Basi di Dati

Prof. Montanari

A.A. 2002/2003

Syllabus dettagliato del corso di

Basi di Dati

Prof. Montanari

A.A. 2002-2003

PARTE 1. Concetti di Base.

Ruolo e funzionalità di una base di dati, astrazione sui dati, modelli dei dati (Entità/Relazioni, relazionale, reticolare, gerarchico e orientato agli oggetti, ibridi relazionali e orientati agli oggetti), istanze e schemi, indipendenza dei dati dal livello logico e fisico, linguaggi per la definizione e la manipolazione dei dati, sistema di gestione di una base di dati (DBMS), amministratore e utenti di una base di dati, il dizionario dei dati, struttura generale di una base di dati.

PARTE 2. Il modello Entità/Relazioni (ER).

Tipi e istanze di entità e di relazioni, attributi (semplici, composti, a singolo valore, multivalore, derivati) e chiavi, dominio di un attributo, vincoli associati alle relazioni (partecipazione e rapporto di cardinalità), relazioni ricorsive e ruoli, relazioni di grado maggiore di 2, entità deboli, gerarchie di specializzazione/generalizzazione (cenni), mapping dal modello ER al modello relazionale, i diagrammi Entità/Relazioni.

PARTE 3. Il Modello Relazionale.

Il modello relazionale e l'algebra relazionale: concetti e vincoli che caratterizzano il modello relazionale dei dati, definizione delle relazioni, operazioni di aggiornamento delle relazioni, l'algebra relazionale, limiti dell'algebra relazionale, operazioni relazionali addizionali, interrogazioni in algebra relazionale, il calcolo relazionale: concetti di base e legame con l'algebra relazionale. SQL e linguaggi host.

PARTE 4. La teoria della progettazione delle basi di dati relazionali.

Dipendenze funzionali, ragionamento sulle dipendenze funzionali, scomposizione di relazioni, scomposizioni lossless-join, scomposizioni che preservano le dipendenze, forme normali per gli schemi di relazione (1NF, 2NF, 3NF e BCNF), scomposizioni lossless-join in BCNF, scomposizioni in 3NF che conservano le dipendenze.

PARTE 5. L'Organizzazione Fisica dei Dati.

Memorizzazione dei record ed organizzazione dei file primari: introduzione, strumenti e tecniche per la gestione della memoria secondaria, buffering dei blocchi, memorizzazione di file di record su disco, operazioni sui file, file di record non ordinati (heap file), file di record ordinati (sorted file), tecniche di hashing, altre possibili organizzazioni dei file primari. Strutture di indicizzazione dei file: tipi di indici ordinati di livello singolo (primari, di clustering, secondari), indici multilivello, indici multilivello dinamici che utilizzano B-alberi e B+-alberi, altri tipi di indici.

PARTE 1

1.1 Introduzione

Una base di dati è una collezione di dati correlati. Per dati si intendono fatti noti che possono essere memorizzati e che hanno un significato implicito. Una base di dati ha le seguenti proprietà implicite:

1. Rappresenta un certo aspetto del mondo reale, talvolta detto il mini-mondo o l'Universo del discorso (UoD: universe of discorse); cambiamenti al mini-mondo si riflettono sulla base di dati;
2. È una collezione di dati logicamente correlati con un certo significato intrinseco; un assortimento casuale di dati non può essere correttamente considerato una base di dati.
3. È progettata, costruita e popolata con dati ad uno scopo specifico; ha uno specifico gruppo di utenti e alcune applicazioni che questi utenti hanno individuato in precedenza essere di loro interesse

In altre parole, una base di dati ha una certa sorgente dalla quale i dati sono derivati, un certo grado di interazione con gli eventi del mondo reale e un pubblico che è attivamente interessato al suo contenuto.
Una base di dati può essere di qualsiasi dimensione e di diversa complessità.
Un sistema di gestione di basi di dati (DBMS: DataBase Management System) è un insieme di programmi che permettono agli utenti di creare e mantenere una base di dati. Il DBMS è perciò un sistema software con scopi generali (general – purpose software system) che facilita il processo di definire, costruire e manipolare una base di dati per varie applicazioni. Definire una base di dati implica specificare i tipi di dati, le loro strutture e i vincoli per i dati che devono essere memorizzati nella base di dati. Costruire la base di dati significa immagazzinare i dati stessi entro un certo mezzo di memorizzazione che è controllato dal DBMS. Manipolare una base di dati include funzioni come la sua interrogazione per recuperare dati specifici, il suo aggiornamento per rispecchiare cambiamenti nel mini-mondo e la generazione di prospetti (reports) a partire dai dati.
Le caratteristiche fondamentali di una base di dati sono:

1. Una BdD ha senso quando bisogna trattare una grossa quantità di dati: *numerosità di dati*. Cioè una quantità di dati che non può stare sulla memoria principale. Una problematica è quindi il trasferimento dei dati dalla memoria secondaria a quella principale. Un fattore di complessità è il numero di trasferimenti dei dati dalla memoria secondaria alla principale. In realtà questo è un problema che nel tempo potrebbe venir superato (ci sarà sempre più memoria principale) ma parallelamente cresce la richiesta di memoria per la BdD (sempre più dati).
2. *Persistenza dei dati*: non ho una situazione di questo tipo quando i dati creati dal programma esistono solo durante l'esecuzione del programma (vengono cancellati con la chiusura del programma). I dati invece devono esistere prima e dopo la durata dell'esecuzione del programma.
3. *Globalità dei dati*: i dati di interesse di un programma sono locali; i dati invece devono essere di interesse per una pluralità di programmi.

Devo quindi costruire un sistema per gestire molti dati persistenti e globali.
Perché non utilizzare un programma? È difficile immaginare un programma che copra tutti gli utilizzi!
Perché non utilizzare il SO (il file system)? In linea di principio è fattibile ma ci sono alcuni problemi:

1. Ridondanza: è difficile controllare che non ci sia ridondanza tra moltissimi dati. Ma la ridondanza potrebbe non essere un problema perché mi costa meno sprecare spazio che fare controlli, ma la ridondanza è spesso finte di inconsistenze dei dati.

2. Difficoltà di accesso ai dati (proliferazione di programmi ad hoc) per ogni problema devo fare un programma apposta; dopo poco tempo ci possono essere moltissimi programmi per fare cose che possono essere molto simili. Si vuole quindi costruire un opportuno linguaggio di interazione con la base di dati con cui fare facilmente tutte le richieste.
3. Disomogeneità dei dati: la struttura dei dati nei diversi file potrebbe essere diversa. Nelle BdD c'è un unico formato dei dati memorizzati.
4. Anomalie dovute da accessi concorrenti ai dati
5. Sicurezza e protezione dei dati: sia sicurezza per problemi hardware e software (cadute di sistema, rotture dei dischi…) sia come garantire che soltanto chi ne ha diritto possa leggere/scrivere i dati.
6. Integrità dei dati: ad esempio come garantire che l'ammontare di un conto non possa andare sotto una soglia prefissata

1.3 Caratteristiche dell'approccio con BdD

1.3.1 Natura autodescrittiva di un sistema di basi di dati

Una caratteristica fondamentale è che il sistema di BdD contiene non solo la base di dati ma anche una definizione o descrizione completa della sua struttura e dei suoi vincoli. Questa definizione è memorizzata nel catalogo di sistema, che contiene informazioni come la struttura di ciascun file, il tipo e il formato di memorizzazione di ciascun dato e vari vincoli sui dati. Le informazioni memorizzate nel catalogo sono dette metadati e descrivono la struttura della base di dati principale. Il catalogo è usato dal software del DBMS e anche dagli utenti della BdD che hanno bisogno di informazioni sulla struttura dela stessa. Un pacchetto software di un DBMS general – purpose non è scritto per una specifica applicazione di BdD, e di conseguenza deve riferirsi al catalogo per conoscere la struttura dei file in un database specifico, come il tipo dei dati a cui accederà. Il software del DBMS deve lavorare egualmente bene con qualsiasi tipo di applicazione di BdD – ad esempio una BdD universitaria, bancaria o aziendale – per tutto il tempo in cui la definizione dei dati permane nel catalogo.
Nella tradizionale gestione file, la definizione di dati è tipicamente parte dei programmi applicativi. Di conseguenza questi programmi sono costretti a lavorare con una BdD specifica, la cui struttura è dichiarata nei programmi applicativi.

1.3.2 Separazione tra programmi e dati e astrazione dei dati

Nella tradizionale gestione dei file la struttura dei file di dati è inserita nei programmi che devono accedervi e pertanto qualsiasi cambiamento alla struttura di un file può richiedere un cambiamento di tutti i programmi che accedono ad esso. Al contrario, i programmi di accesso del DBMS nella maggioranza dei casi non richiedono cambiamenti di questo tipo. La struttura dei file di dati è memorizzata nel catalogo del DBMS separatamente dai programmi di accesso. Questa proprietà è chiamata indipendenza di programmi e dati.
Nelle BdD orientate agli oggetti e in quelle relazionali gli utenti possono definire le operazioni sui dati come parte delle definizioni delle BdD. Un'operazione (detta anche funzione) è specificata in due parti: l'interfaccia e il metodo. I programmi applicativi dell'utente possono operare sui dati invocando queste operazioni attraverso i loro nomi e argomenti, indipendentemente da come le operazioni stesse siano implementate. Questo fatto può essere definito indipendenza tra programmi e operazioni.
La caratteristica che consente l'indipendenza tra programmi e dati e l'indipendenza tra programmi e operazioni è detta astrazione sui dati. Un DBMS fornisce agli utenti una rappresentazione concettuale dei dati che non comprende molti dettagli su come i dati sono memorizzati e su come le operazioni sono implementate. Informalmente un modello di dati è un tipo di astrazione dei dati usato per fornire questa rappresentazione concettuale. Il modello di dati usa concetti logici come

oggetti, le loro proprietà e le loro interrelazioni, che sono per molti utenti più facili da capire rispetto a nozioni di memorizzazione nel calcolatore.
Nell'approccio con BdD la struttura e l'organizzazione dettagliate di ciascun file sono memorizzate nel catalogo. Gli utenti fanno riferimento alla rappresentazione concettuale dei file, e il DBMS estrae dal catalogo i dettagli della loro memorizzazione quando questi siano necessari al software del DBMS, Molti modelli di dati diversi possono essere usati per fornire tale astrazione dei dati agli utenti di BdD. Con la recente tendenza verso BdD orientate agli oggetti e BdD relazionali, l'astrazione è portata a un livello superiore per includere non solo la struttura dei dati ma anche le operazioni sui dati. Queste operazioni forniscono un'astrazione di attività del mini – mondo comunemente comprensibili agli utenti. In questo senso, un'astrazione dell'attività del mini – mondo è resa disponibile all'utente come operazione astratta.

1.3.3 Supporto di viste multiple dei dati

Tipicamente una BdD ha molti utenti, ciascuno dei quali può richiederne una diversa prospettiva o vista. Una vista è un sottoinsieme della BdD o può contenere dati virtuali che sono derivati dai file della BdD ma non sono esplicitamente memorizzati.

1.3.4 Condivisione dei dati ed elaborazione delle transazioni con utenti multipli.

Un DBMS multiutente deve consentire a più utenti di accedere contemporaneamente alla base di dati. La cosa è essenziale se dati per applicazioni multiple devono essere integrati e mantenuti in una singola BdD. Il DBMS deve contenere un software per il controllo della concorrenza che garantisca che più utenti che cerchino di aggiornare gli stessi dati lo possano fare in maniera controllata, cosicché il risultato degli aggiornamenti sia corretto. Applicazioni di questo tipo sono generalmente dette applicazioni di elaborazione delle transazioni in linea (OLTP: On-Line Transaction Processing). Una funzione fondamentale del software di un DBMS multiutente è quella di assicurare che le transazioni concorrenti operino correttamente.

1.4 Gli attori in scena

1.4.1 Il gestore di una base di dati

Il gestore di una BdD (database manager) è un insieme di moduli software che realizza l'interfaccia con i dati fisicamente presenti nella BdD (low-level data) che sono essenzialmente trattati dal file system. Esso supporta le seguenti funzionalità:

- Interazione con il file system
- Gestione dei vincoli di integrità
- Gestione della sicurezza dei dati
- Backup e recovery
- Controllo della concorrenza.

1.4.2 l'amministratore di una base di dati

L'amministratore di una base di dati (DBA: database administrator) è il responsabile dei dati e delle relative procedure di accesso. Egli svolge le seguenti funzioni:

- Definizione dello schema;
- Definizione delle strutture di memorizzazione e dei relativi metodi di accesso;
- Eventuale modifica dello schema logico e/o dell'organizzazione fisica dei dati;
- Assegnazione delle autorizzazioni per l'accesso ai dati;
- Specifica dei vincoli di integrità.

1.4.3 Utenti di una base di dati

Possiamo distinguere più classi di utenti di una BdD sulla base delle diverse possibili modalità di interazione col sistema:

- Utenti naive
- Utenti che sviluppano programmi applicativi scritti in un linguaggio di alto livello, tipo Cobol, Pascal, C, che necessitano di accedere alla BdD (uso del DML all'interno di linguaggio host);
- Utenti sofisticati, che utilizzano direttamente il DML.

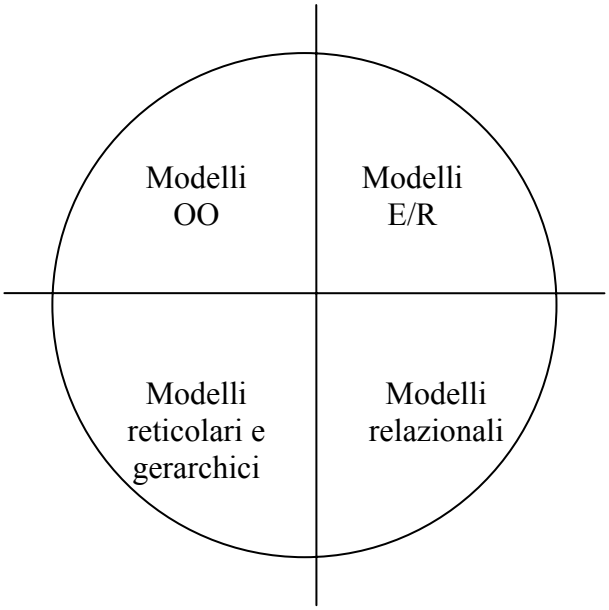
2.1 Modelli di dati, schemi e istanze.

Una caratteristica fondamentale dell'approccio con BdD è che esso fornisce un certo livello di astrazione dei dati nascondendo quei dettagli sulla memorizzazione degli stessi che non sono necessari alla maggior parte degli utenti. Un modello di dati frisane i mezzi necessari per raggiungere questa astrazione. Per struttura di un BdD si intendono i tipi di dati, le associazioni e i vincoli che dovrebbero valere su di essi. La maggior parte dei modelli di dati comprende pure un insieme di operazioni di base per specificare recuperi e aggiornamenti sulla BdD.
In aggiunta alle operazioni di base fornite dal modello di dati sta diventando sempre più comune includere in esso concetti per specificare l'aspetto dinamico o il comportamento di un'applicazione di BdD. Ciò consente al progettista della BdD di specificare un insieme di operazioni definita dall'utente valide, permesse sugli oggetti della BdD. Un esempio potrebbe essere una funzione "calcola_media_voti" che può essere applicata ad ogni oggetto di tipo studente. D'altra parte, operazioni generiche per inserire, cancellare, modificare e recuperare qualsiasi tipo di oggetto sono spesso comprese nelle operazioni di base del modello di dati.

2.1.1 Categorie di modelli di dati

I modelli di dati di alto livello o concettuali forniscono concetti che sono vicini al modo in cui utenti percepiscono i dati, mentre i modelli di dati di basso livello o fisici forniscono concetti che descrivono i dettagli sul modo in cui i dati sono memorizzati sul computer. I concetti forniti dai modelli di dati di basso livello sono generalmente destinati a specialisti di computer, non a utenti finali tipici. Tra questi due estremi c'è una classe di modelli di dati implementabili che forniscono concetti che possono essere compresi dagli utenti finali ma che non sono troppo lontani dal modo in cui i dati sono organizzati entro il computer: essi nascondono alcuni dettagli di memorizzazione dei dati ma proprio perché implementabili si possono considerare più vicini al computer stesso.
Un modello dei dati è un insieme di strumenti concettuali utilizzati per descrivere i dati, le relazioni (i legami) tra i dati, la semantica dei dati e gli eventuali vincoli sui dati.
I modelli di dati concettuali usano concetti come entità, attributi e associazioni. Un'entità rappresenta un oggetto o concetto del mondo reale, come un impiegato o un progetto, che è descritto nella base di dati. Un attributo rappresenta una qualche proprietà di interesse che descrive più a fondo un'entità, come il nome o il salario di un impiegato. Un'associazione tra due o più entità rappresenta un'interazione tra le entità; per esempio un'associazione lavorativa tra impiegato e progetto.
Quali sono questi modelli? In ordine cronologico:

1. Reticolari e gerarchici ('60)
2. Relazionale ('70): a tutt'oggi è il modello di riferimento
3. Entità/Relazione ('80)
4. Orientato agli oggi ('80): ciò che si ottiene unendo la programmazione orientata agli oggetti e la base di dati. È necessario per alcune classi di applicazioni che servono per dati molto complessi.
5. Ibridi relazionali/OO ('90): le caratteristiche tipiche dei modelli orientati agli oggetti vengono aggiunti nei modelli relazionali.



Modelli basati sugli oggetti: vedono una BdD come una collezione di ogetti che possono essere definiti in termini di altri oggetti (non so lo spazio di memoria che servirà).

Modelli basati sui record: la mia BdD è una collezione di record di diverso tipo di dimensione prefissata (ciascun record è composta da campi e ogni campo deve contenere dati di dimensione fissa

Modelli basati sull'identità degli oggetti: a ciascun oggetto è associato un identificatore univoco

Modelli basati sui valori degli oggetti: ogni oggetto è identificato univocamente dal valore.

2.1.2 Schemi, istanze e stato di una BdD

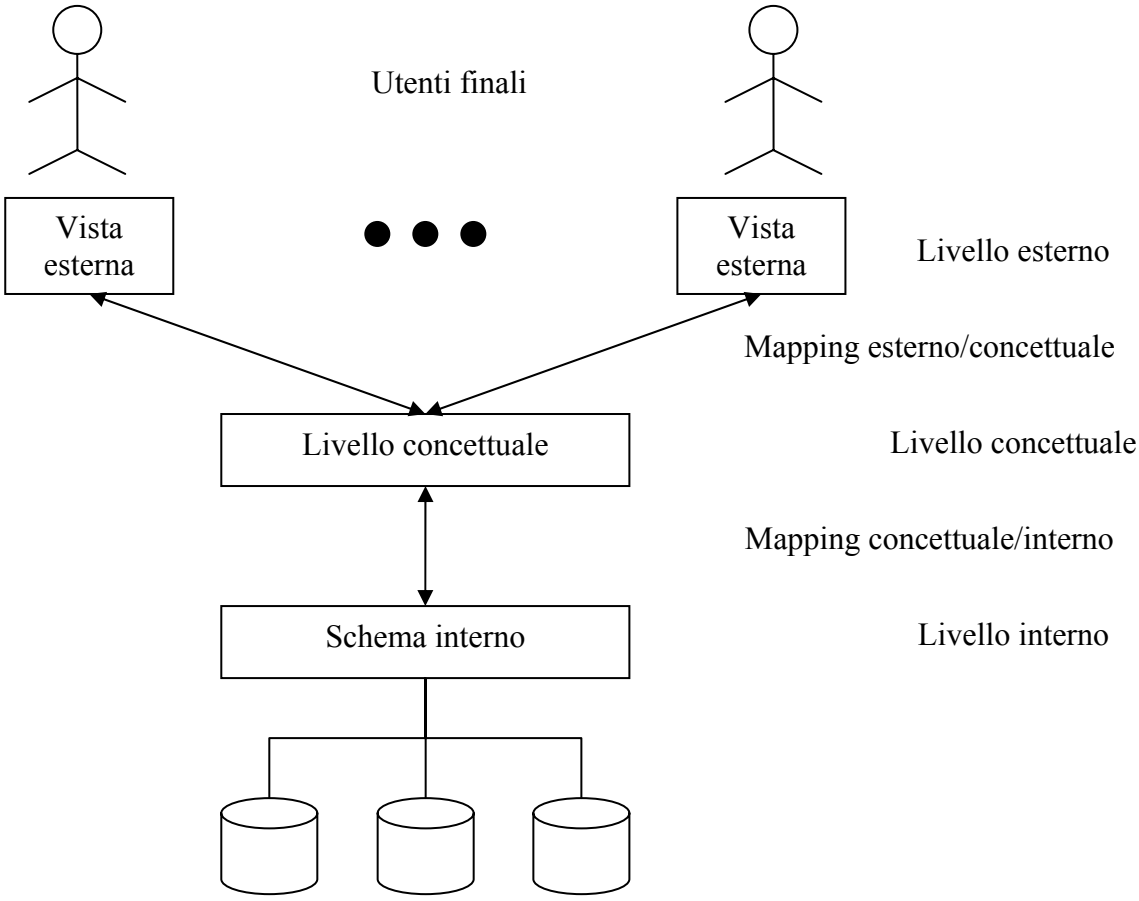
È importante distinguere tra la descrizione della BdD e la BdD stessa. La descrizione è detta schema della base di dati (data-base schema); esso è specificato durante la progettazione della base di dati e non ci si aspetta che cambi frequentemente. Una rappresentazione grafica di uno schema è detta diagramma di schema. Si chiama ciascun oggetto dello schema – come studente o insegnamento – costrutto dello schema.
Un diagramma di schema rappresenta solo certi aspetti di uno schema, come i nomi dei vari tipi di record e dei dati, e alcuni tipi di vincoli; altri aspetti non vengono specificati. Molti tipi di vincoli non sono rappresentati nei diagrammi di schema; per esempio, un vincolo come "gli studenti che stanno frequentando Informatica devono superare l'esame CS1310 prima della fine del secondo anno di corso" è piuttosto difficile da rappresentare.
I dati nella base di dati in un particolare istante di tempo costituiscono lo stato della BdD o l'istantanea della BdD. Si parla anche di insieme di occorrenze o istante nella BdD. In certo stato della BdD ciascun costrutto Studente conterrà l'insieme di entità (record) di ciascuno studente come sue istanze.
Quando si definisce una nuova BdD, si specifica il suo schema solo al DBMS. A questo punto il corrispondente stato della base di dati è lo stato vuoto, senza dati. Si raggiunge lo stato iniziale della base di dati quando essa è per la prima vola popolata o caricata con i dati iniziali.

2.2 Architettura di un DBMS e indipendenza dei dati.

2.2.1 L'architettura a tre livelli

L'obiettivo dell'architettura a tre livelli è quello di separare le applicazioni dell'utente dalla base di dati fisica. In questa architettura possono essere definiti schemi ai seguenti tre livelli:

1. Il livello interno ha uno schema interno, che descrive la struttura di memorizzazione fisica della base di dati. Lo schema interno usa un modello fisico di dati e descrive i dettagli completi della memorizzazione dei dati e dei cammini di accesso per la base di dati.
2. il livello concettuale ha uno schema concettuale, che descrive la struttura dell'intera BdD per una comunità di utenti. Lo schema concettuale nasconde i dettagli delle strutture di memorizzazione fisica e si concentra sulla descrizione di entità, tipi di dati, associazioni, operazioni degli utenti e vincoli. A questo livello può essere usato un modello di dati di alto livello o un modello di dati implementabile.
3. Il livello esterno o di vista comprende un certo numero di schemi esterni o viste d'utente. Ciascuno schema esterno descrive la parte della BdD a cui è interessato un particolare gruppo di utenti, nascondendo il resto. A questo livello possono essere usati un modello di dati di alto livello o un modello di dati implementabile.



2.2.2 Indipendenza dei dati

L'architettura a tre livelli può essere usata per spiegare il concetto di indipendenza dei dati che può essere definita come la capacità di cambiare lo schema a un certo livello di un sistema di BdD senza dover cambiare lo schema al livello immediatamente superiore. È possibile definire due tipi di indipendenza dei dati:

1. *Indipendenza logica dei dati*: è la capacità di cambiare lo schema concettuale senza dover cambiare gli schemi esterni o i programmi applicativi. Si può cambiare lo schema concettuale per espandere la base di dati (aggiungendo un tipo di record o un campo) o ridurla (anche se in quest'ultimo caso non dovrebbero essere intaccati solo gli schemi che si riferiscono ai dati rimanenti).
2. *Indipendenza fisica dei dati*: è la capacità di cambiare lo schema interno senza dover cambiare lo schema concettuale. Cambiamenti allo schema interno possono essere necessari perché alcuni file fisici devono essere riorganizzati.

2.3 Linguaggi e interfacce di BdD

2.3.1 Linguaggi dei DBMS

In molti DBMS nei quali non è mantenuta una separazione netta tra i due livelli, un solo linguaggio, detto linguaggio di definizione dei dati (DDL: Data Definition Language), è usato dal DBA e dai progettisti per definire gli schemi. Il DBMS avrà un compilatore DDL la cui funzione sarà quella di elaborare le istruzioni DDL per identificare descrizioni dei costrutti dello schema e per memorizzare la descrizione dello schema nel catalogo del DBMS.
In quei DBMS in cui è mantenuta una netta distinzione tra i livelli concettuale e interno, il DDL è usato per specificare il solo schema concettuale. Un altro linguaggio, il linguaggio di definizione della memorizzazione (SDL: Storage Definition Language), è usato per specificare lo schema interno. Per una vera architettura a tre livelli si avrà bisogno di un terzo linguaggio, il linguaggio di definizione delle viste (VDL: View Definition Language), per specificare le viste d'utente e i loro mapping nello schema concettuale.
Una volta che gli schemi sono stati compilati e la base di dati è stata popolata, gli utenti devono avere gli strumenti per manipolare quest'ultima. Le manipolazioni tipiche comprendono il recupero, l'inserimento, la cancellazione e la modifica dei dati. Il DBMS fornisce un linguaggio di manipolazione dei dati (DML: Data Manipulation language) per questi scopi.
Ci sono due tipi principali di DML. Un DML di alto livello o nonprocedurale può essere usato da solo per specificare in modo conciso operazioni complesse sulle basi di dati. Molti DBMS consentono che istruzioni DML di alto livello vengano inserite interattivamente da un terminale o vengono incapsulate in un linguaggio di programmazione. In quest'ultimo caso le istruzioni DML devono essere riconosciute all'interno del programma così da poter essere estratte da un precompilatore ed elaborate dal DBMS. Un DML di basso livello o procedurale deve essere incapsulato in un linguaggio di programmazione. Questo tipo di DML tipicamente recupera singoli record od oggetti della base di dati e li elabora separatamente uno ad uno. Ha quindi bisogno di usare costrutti per linguaggi di programmazione, come l'iterazione, per recuperare ed elaborare ciascun record, prendendolo da un insieme di record.

PARTE 3

7. Il modello di dati, i vincoli e l'algebra relazionali

7.1 Concetti del mondo relazionale

Il modello relazionale rappresenta la base di dati come una collezione di relazioni. Informalmente, ogni relazione assomiglia a una tabella di valori o, per certi versi, a un file di record "piatto". Ci sono però importanti differenze tra relazioni e file.
Quando si pensa a una relazione come a una tabella di valori, ogni riga della tabella rappresenta una collezione di valori di dati collegati. Nel modello relazionale ogni riga della tabella rappresenta un fatto che tipicamente corrisponde ad un'entità o a un'associazione del mondo reale. Il nome della tabella e i nomi delle colonne sono usati per aiutare ad interpretare il significato dei valori presenti in ogni riga.
Nella terminologia formale del modella relazionale, una riga è detta tupla, un'intestazione di colonna è detta attributo e la tabella è detta relazione. Il tipo di dati che descrive tipi di valori che possono apparire in ogni colonna è detto dominio.

7.1.1 Domini, attributi, tuple e relazioni.

Un dominio D è un insieme di valori atomici. Per atomici si intende che ogni valore del dominio è indivisibile , per lo meno per quanto riguarda il modello relazionale. Un metodo comune per specificare un dominio consiste nello specificare un tipo di dati, da cui sono tratti i valori dei dati che formano il dominio. È anche utile specificare un nome per il dominio, per aiutare ad interpretare i suoi valori (ad es. Numeri_Telefonici_USA: l'insieme di numeri telefonici a 10 cifre validi negli Stati Uniti).
Queste definizioni di domini sono dette logiche. Per ogni dominio viene anche specificato un tipo di dati o formato. Ad esempio il tipo di dati per i numeri_telefonici_USA può essere dichiarato come una stringa di caratteri (ddd)ddd-dddd dove ogni d è un numero intero decimale.
Uno schema di relazione R, indicato con $R(A_1, A_2, \ldots, A_n)$, è costituito da un nome di relazione R e da un elenco di attributi $A_1, A_2, \ldots, A_n$. Ogni attributo A_i fornisce il nome del ruolo interpretato da un certo dominio D nello schema di relazione R. D è detto dominio di A_i,ed è indicato con $dom(A_i)$.
Uno schema di relazione è usato per descrivere una relazione; R è detto nome della relazione. Il grado di una relazione è il numero di attributi presenti nel suo schema di relazione.
Un esempio di uno schema di relazione per una relazione di grado 7 che descrive studenti universitari è il seguente:

STUDENTE(Nome, SSN, TelefonoDiCasa, Indirizzo, TelefonoUfficio, Età, MV)

Una relazione (o stato di relazione) r dello schema di relazione $R(A_1, A_2, \ldots, A_n)$, indicata anche con r(R), è un insieme di n-tuple $r=\{t_1, t_2, \ldots, t_n\}$. Ogni n-tupla t è un elenco ordinato di n valori $t = < v_1, v_2, \ldots, v_n >$, dove ogni valore v_i, $1 \leq i \leq n$, è un elemento di $dom(A_i)$ oppure è uno speciale valore null. Il valore i-esimo nella tupla t, che corrisponde all'attributo A_i, è indicato anche con $t[A_i]$.
La relazione è rappresentata graficamente come una tabella, in cui ogni tupla è rappresentata con una riga e ogni attributo corrisponde a un'intestazione di colonna che indica un ruolo o un'interpretazione dei valori presenti in quella colonna. I valori nulli rappresentano attributi i cui valori sono sconosciuti o non esistono per alcune singole tuple.
La definizione sopra data di relazione può essere riformulata nel modo seguente:. Una relazione r(R) è una relazione matematica di grado n sui domini $dom(A_1)$, $dom(A_2)$,… , $dom(A_n)$, cioè è un sottoinsieme del prodotto cartesiano dei domini che definiscono R:

$$r(R) \subseteq (dom(A_1) \text{ X } dom(A_2) \text{ X} \ldots \text{ X } dom(A_n))$$

Il prodotto cartesiano specifica tutte le possibili combinazioni di valori dei domini sottostanti. Perciò, se si indica con |D| il numero dei valori o cardinalità di un dominio D, e supponiamo che tutti i domini siano finiti, il numero totale di tuple del prodotto cartesiano è

$$|dom(A_1)| * |dom(A_2)| *\ldots * |dom(A_n)|$$

7.1.2 Caratteristiche delle relazioni

Ordinamento delle tuple in una relazione: una relazione è definita come un insieme di tuple, ma ordinato. L'ordinamento delle tuple in realtà non fa parte della definizione di una relazione, perché una relazione tenta di rappresentare fatti a livello logico o astratto.
Ordinamento dei valori all'interno di una tupla e definizione alternativa di relazione: Secondo la definizione precedente di relazione, una n-tupla è un elenco ordinato di n valori, cosicché l'ordinamento dei valori in una tupla – e perciò gli attributi nella definizione di uno schema di relazione – è importante. Comunque, a livello logico, l'ordine degli attributi e dei loro valori non è importante, purché sia mantenuta la corrispondenza tra attributi e valori.
Valori nelle tuple: ogni valore in una tupla è un valore atomico, cioè non è divisibile in parti componenti, usando una struttura del modello relazionale di base. Perciò gli attributi composti e multivalore non sono consentiti (assunzione di prima forma normale).
I valori di alcuni attributi all'interno di una specifica tupla possono essere sconosciuti o non pertinenti per quella tupla. In questi casi si usa il valore null.
Interpretazione di una relazione: lo schema di relazione può essere interpretato come una dichiarazione, ossia un tipo di asserzione. Ogni tupla nella relazione può perciò essere interpretata come un fatto o una particolare istanza dell'asserzione.

7.1.3 Notazione del modello relazionale

- Uno schema di relazione R di grado n è indicato con $R(A_1, A_2, \ldots, A_n)$
- Una n-tupla di una relazione r(R) è indicata con $t = < v_1, v_2, \ldots, v_n >$, dove v_i è il valore corrispondente all'attributo A_i. La notazione seguente si riferisce ai valori componenti delle tuple:
    a. Sia $t[A_i]$ sia $t.A_i$ si riferiscono al valore v_i di t per l'attributo A_i.
    b. Sia $t[A_u, A_{w, \ldots,} A_z]$ sia $t.(A_u, A_{w, \ldots,} A_z)$ dove $A_u, A_{w, \ldots,} A_z$ è una lista di attributi di R, si riferiscono alla sottotupla di t di valori $< v_u, v_{w, \ldots,} V_z >$ corrispondenti agli attributi specificati nella lista
- Le lettere Q, R ed S indicano nomi di relazione
- Le lettere q, r, s indicano stati di relazioni;
- Le lettere t,u,v indicano tuple

7.2 Vincoli relazioni e schemi di BdD relazionali

7.2.1 Vincoli sul dominio

I vincoli sul dominio stabiliscono che il valore di ogni attributo A deve essere un valore atomico del dominio dom(A).

7.2.2 Vincoli di chiave e vincoli sui valori nulli.

Una relazione è definita come un insieme di tuple. Tutti gli elementi di un insieme sono distinti per definizione; perciò anche tutte le tuple di una relazione devono essere distinte. Ciò significa che nessuna coppia di tuple può assumere la stessa combinazione di valori per tutti gli attributi. Di solito, ci sono altri sottoinsiemi di attributi di uno schema di relazione R con la proprietà che nessuna coppia di tuple possa avere la stessa combinazione di valori per questi attributi. Si supponga di denotare questo sottoinsieme di attributi SK, allora

$$t_1[SK] \neq t_2[SK]$$

Ogni insieme SK di attributi di questo tipo è detto superchiave dello schema di relazione R. Una superchiave SK definisce un vincolo di univocità secondo il quale nessuna coppia di tuple distinte in uno stato r di R può avere lo stesso valore per SK. Ogni relazione ha almeno una superchiave: l'insieme di tutti i suoi attributi. Però una superchiave può avere attributi ridondanti, cosicché un concetto più utile è quello di chiave, che non ha ridondanza. Una chiave (key) K di uno schema di relazione R è una superchiave di R, con la proprietà aggiuntiva che la rimozione di qualsiasi attributo A da K porta ad un insieme di attributi K' che non è superchiave di R. Perciò una chiave è una superchiave minimale.

7.2.3 BdD e schemi di BdD relazionali

Uno schema di BdD relazionale S è costituito da un insieme di schemi di relazione S = $\{R_1, R_2, \ldots, R_m\}$ e da un insieme di vincoli di integrità IC (Integrity Constrains). Uno stato di base di dati relazionale di S è costituito da un insieme di stati di relazione DB=$\{r_1, r_2, \ldots, r_m\}$ tale che ogni r_i sia uno stato di R_i e che gli stati di relazione r_i soddisfino i vincoli di integrità specificati in IC.

7.2.4 Integrità dell'entità, integrità referenziale e chiavi esterne

Il vincolo di integrità dell'entità stabilisce che nessun valore di chiave primaria può essere nullo.
Il vincolo di integrità referenziale è specificato tra sue relazioni ed è usato per mantenere la consistenza fra tuple di due relazioni. Informalmente il vincolo di integrità referenziale stabilisce che una tupla di una relazione che fa riferimento a un'altra relazione deve fare riferimento ad una tupla esistente in quella relazione.
Per definire più formalmente l'integrità referenziale, definiamo dapprima il concetto di chiave esterna. Le condizioni di chiave esterna specificano un vincolo di integrità referenziale tra i due schemi di relazione R_1 ed R_2. Un insieme di attributi FK (foreign key) nello schema di relazione R_1 è una chiave esterna di R_1 che riferisce la relazione R_2 se soddisfa le seguenti due regole:

1. Gli attributi presenti in FK hanno gli stessi domini degli attributi di chiave primaria PK (Primary Key) di R_2; si dice che gli attributi FK riferiscono a fanno riferimento alla relazione R_2.
2. Un valore di FK in una tupla t_1 dello stato corrente $r_1(R_1)$ o è presente come valore di PK in una qualche tupla t_2 nello stato corrente $r_2(R_2)$ o è nullo. Nel primo caso si ha $t_1[FK]=t_2[PK]$ e si dice che la tupla t_1 fa riferimento o riferisce alla tupla t_2.

7.3 Operazioni di aggiornamento e gestione delle violazioni dei vincoli

Le operazioni del modello relazionale possono essere classificate in operazioni di recupero (retrievals) e di aggiornamento (updates). Esistono tre operazioni fondamentali di aggiornamento sulle relazioni: (1) inserimento, (2) cancellazione e (3) modifica.

7.3.1 L'operazione di inserimento

L'operazione di inserimento fornisce un elenco di valori di attributi per una nuova tupla t che deve essere inserita in una relazione R. L'inserimento può violare tutti e quattro i tipi di vincoli. I vincoli sul dominio possono essere violati se viene fornito un valore di attributo che non compare nel dominio corrispondente. I vincoli di chiave possono essere violati se il valore di una chiave nella tupla esiste già in un'altra tupla della relazione r(R). L'integrità dell'entità può essere violata se la chiave primaria della nuova tupla t ha valore nullo. L'integrità referenziale può essere violata se il valore di una qualche chiave esterna di t fa riferimento a una tupla che nella relazione riferita non esiste.
Se un inserimento viola uno o più vincoli l'opzione di default consiste nel rifiutare l'inserimento.

7.3.2 L'operazione di cancellazione

L'operazione di cancellazione (dolete) può violare solo l'integrità referenziale, il che avviene se la tupla che viene cancellata è riferita dalle chiavi esterne di altre tuple della BdD.
Se un'operazione di cancellazione causa una violazione sono disponibili tre opzioni: la prima consiste nel rifiutare al cancellazione. La seconda consiste nel tentare di far ricadere a cascata (ossia propagare), cancellando tutte le tuple che si riferiscono alla tupla che si sta cancellando. Una terza opzione consiste nel modificare i valori degli attributi riferenti che causano la violazione, ogni valore di questo tipo è posto a null oppure cambiato per riferire un'altra tupla valida. Si noti che, se un attributo riferente che causa una violazione fa parte della chiave primaria, esso non può essere posto uguale al valore nullo, dato che altrimenti violerebbe l'integrità dell'entità.

7.3.3 L'operazione di modifica

L'operazione di modifica è usata per cambiare i valori di uno o più attributi di tuple presenti in una qualche relazione R.
Di solito la modifica di una attributo che non è né una chiave primaria né una chiave esterna non dà luogo a nessun problema. Il DBMS deve solo verificare se il nuovo valore è del tipo di dati e del dominio corretti. La modifica del valore di una chiave primaria è simile alla cancellazione di una tupla e all'inserimento di un'altra al suo posto, dato che viene usata la chiave primaria per identificare le tuple. Entrano quindi in gioco i problemi esaminati prima per l'inserimento e la cancellazione. Se viene modificato un attributo di chiave esterna, ilo DBMS deve sincerarsi che il nuovo valore faccia riferimento a una tupla presente nella relazione riferita (oppure sia nullo).

7.4 Operazioni di base dell'algebra relazionale

Oltre a definire la struttura e i vincoli della BdD, un modello di dati deve contenere un insieme di operazioni per manipolare i dati. Un insieme di base di operazioni del modello relazionale costituisce l'algebra relazionale. Queste operazioni consentono all'utente di specificare le richieste di recupero fondamentali. Il risultato del recupero è una nuova relazione, che può essere stata formata a partire da una o più relazioni. Le operazioni dell'algebra, perciò, producono nuove relazioni, le quali possono essere ulteriormente manipolate usando operazioni della stessa algebra. Una sequenza di operazioni di algebra relazionale forma un'espressione dell'algebra relazionale, il cui risultato è ancora una relazione.
Le operazioni dell'algebra relazionale sono generalmente divise in sue gruppi. Un gruppo comprende le operazioni insiemistiche come unione, intersezione, differenza e prodotto cartesiano. L'altro gruppo è composto da operazioni specificate sviluppate per le basi di dati relazionali; esse comprendono selezione, proiezione e join.

7.4.1 L'operatore di selezione.

L'operazione di selezione è usata per selezionare un sottoinsieme di tuple di una relazione che soddisfa una condizione di selezione. Si può considerare tale operazione come un filtro che trattiene solo quelle tuple che soddisfano una condizione qualificante. In generale l'operazione è indicata con:

$$\sigma_{<\text{condizione di selezione}>}(R)$$

Dove il simbolo il simbolo σ (sigma) è usato per denotare l'operatore di selezione, e la condizione di selezione è un'espressione booleana specificata sugli attributi della relazione R. Si noti che generalmente un'espressione dell'algebra relazionale il cui risultato è una relazione; l'espressione più semplice è costituita soltanto dal nome di una relazione della BdD. La relazione risultante dall'opeazione di selezione ha gli stessi attributi di R. L'espressione booleana specificata in <condizione di selezione> è costituita da un certo numero di clausole della forma:

<nome di attributo><op di confronto><valore costante> o
<nome di attributo><op di confronto><nome di attributo>

Dove <nome di attributo> è il nome di una ttributo di R, <op di confronto> è di solito uno degli operatori {=, ≠, <, ≤, >, ≥}, e <valore costante> è un valore costante del dominio dell'attributo. Le clausole possono essere unite arbitrariamente tramite gli operatori AND, OR e NOT per formare una condizione di selezione generale.
Si noti che gli operatori di confronto nell'insieme {=, ≠, <, ≤, >, ≥} si applicano ad attributi i cui domini sono costituiti da valori ordinati, come ad esempio domini numerici o date. Nel caso in cui il dominio dell'attributo è costituito da valori non ordinati (p.es. {rosso, verde, marrone, giallo}) si possono usare solo gli operatori {=, ≠}.
In generale, il risultato di un'operazione di selezione può essere determinato come segue. La <condizione di selezione> è applicata indipendentemente a ciascuna tupla t di R, sostituendo ogni occorrenza di un attributo A, nella condizione di selezione con il suo valore nella tupla $t[A_i]$. Se la condizione viene valutata vera, allora la tupla t viene selezionata. Nel risultato dell'operazione di selezione sono presenti tutte le tuple selezionate. Le condizioni booleane AND, OR e NOT sono interpretate nel solito modo.
Inoltre l'operazione di selezione si applica a ciascuna tupla individualmente; perciò le condizioni di selezione non possono coinvolgere più di una tupla. Il grado della relazione risultante da un'operazione di selezione è uguale a quello di R. Il numero di tuple nella relazione risultante è sempre minore o uguale al numero di tuple presenti in R. Ci si riferisce alla frazione di tuple selezionate tramite una condizione di selezione come una selettività della condizione.
Si noti che l'operazione di selezione è commutativa, ossia

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

una sequenza di operazioni di selezione può perciò essere eseguita in ogni ordine. Inoltre è sempre possibile unire una cascata di operazioni di selezione in una sole operazione con una condizione di congiunzione AND:

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ldots)) = \sigma_{<cond1>\text{AND}<cond2>\text{ AND}\ldots\text{AND}<condn>}(R)$$

7.4.2 L'operazione di proiezione

Se si pensa a una relazione come a una tabella, l'operazione di selezione seleziona alcune delle righe di una tabella scartandone altre. L'operazione di proiezione, d'altro lato, seleziona certe colonne della tabella e scarta altre. Se si è interessati solo a certi attributi di una relazione, si può usare l'attributo di proiezione per proiettare la relazione solo su di essi.
La forma generale dell'operazione di proiezione è:

$$\pi_{<\text{lista di attributi}>}(R)$$

dove π è il simbolo usato per rappresentare l'operazione e <lista di attributi> è un elenco di atributi, presi fra quelli della relazione R. Si noti anche che R, in generale è un'espressione di algebra relazionale il cui risultato è una relazione; nel caso più semplice è proprio il nome di una relazione della BdD. Il risultato dell'operazione di proiezione ha solo gli attributi specificati nella <lista di attributi>, e nello stesso ordine con cui compaiono nella lista. Perciò il suo grado è uguale al numero di attributi in <lista di attributi>.
Se la lista di attributi comprende solo attributi non chiave di R, è probabile che se presentino tuple duplicate; l'operatore di proiezione rimuove tutte le tuple duplicate, in modo che il risultato sia un insieme di tuple e perciò una relazione valida: ciò è noto come eliminazione dei duplicati.
Il numero di tuple di una relazione risultante a partire da un'operazione di proiezione è sempre minore o uguale al numero di tuple presenti in R.

7.4.3 Sequenze di operazioni e operazione di ridenominazione

È possibile scrivere le operazioni come una singola espressione dell'algebra relazionale, nidificando le operazioni, oppure applicare una operazione alla volta e creare relazioni che danno il risultato intermedio. In quest'ultimo caso occorre dare un nome alle relazioni che contengono i risultati intermedi. Ad esempio:

$$\pi_{\text{NOME_BATT, COGNOME, STIPENDIO}}(\sigma_{\text{N_D=5}}(\text{IMPIEGATO}))$$

si può riscrivere come

$$IMP_DIP5 \leftarrow \sigma_{N_D=5}(IMPIEGATO)$$
$$RISULTATO \leftarrow \pi_{NOME_BATT,\ COGNOME,\ STIPENDIO}(IMP_DIP5)$$

Si possono anche ridenominare gli attributi:

$$IMP_DIP5 \leftarrow \sigma_{N_D=5}(IMPIEGATO)$$
$$R(NOME_DI_BATTESIMO, NOME_DI_FAMIGLIA, STIPENDIO) \leftarrow \pi_{NOME_BATT,\ COGNOME,\ STIPENDIO}(IMP_DIP5)$$

7.4.4 Operazioni insiemistiche

Il gruppo successivo di operazioni di algebra relazionale è costituito dalle usuali operazioni matematiche sugli insiemi. Ad esempio, per recuperare i numeri di previdenza sociale di tutti gli impiegati che o lavorano nel dipartimento 5 o sorvegliano direttamente un impiegato che lavora nel dipartimento 5, si può usare l'operazione di UNIONE nel modo seguente:

$$IMP_DIP5 \leftarrow \sigma_{N_D=5}(IMPIEGATO)$$
$$RISULTATO1 \leftarrow \pi_{SSN}\ (IMP_DIP5)$$
$$RISULTATO2(SSN) \leftarrow \pi_{SUPERSSN}\ (IMP_DIP5)$$
$$RISULTATO \leftarrow RISULTATO1 \cup RISULTATO2$$

L'operazione di UNIONE fornisce le tuple che sono o in RISULTATO1 o in RISULTATO2 o in entrambi. Molte operazioni sugli insiemi vengono usate per fondere gli elementi di due insiemi in modi diversi; fra queste, in particolare, l'unione, l'intersezione e la differenza. Queste operazioni devono essere eseguite su relazioni che hanno lo stesso tipo di tuple; questa condizione è detta compatibilità all'unione. Si dice che due relazioni $R(A_1, A_2, \ldots, A_n)$ e $S(B_1, B_2, \ldots, B_n)$ sono compatibili all'unione se hanno lo stesso grado e se $dom(A_i) = dom(B_i)$ per $1 \leq i \leq n$.
È possibile definire queste tre operazioni su due relazioni R e S compatibili all'unione nel modo seguente:

- *Unione*: il risultato di questa operazione, indicato con $R \cup S$, è una relazione che comprende tutte le tuple che sono in R o in S o sia in R sia in S. Le tuple duplicate vengono eliminate.
- *Intersezione*: il risultato di questa operazione, indicato con $R \cap S$, è una relazione che comprende tutte le tuple che sono sia in R sia in S.
- *Differenza*: il risultato di questa operazione, indicato con R – S, è una relazione che comprende tutte le tuple che sono in R ma non in S.

Si consideri ora l'operazione di prodotto cartesiano – noto anche come prodotto incrociato (cross product) o join incrociato (cross join) – indicato con x; si tratta ancora di un'operazione insiemistica binaria, ma le relazioni alle quali si applica non devono necessariamente essere compatibili al'uonione. Questa operazione è utilizzata per unire tuple prese da due relazioni in modo combinatorio. In generale, il risultato di $R(A_1, A_2, \ldots, A_n)$ x $S(B_1, B_2, \ldots, B_m)$ è una relazione Q con n+m attributi $R(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, presi in quest'ordine. La relazione risultante Q presenta una tupla per ogni combinazione di tuple – una di R e una di S. Perciò, se R ha n tuple e S ne ha m, allora R x S avrà n * m tuple. L'operazione applicata da sola è generalmente priva di significato. È invece utile quando è seguita da una selezione che accoppia i valori degli attributi provenienti dalle relazioni componenti.
Il prodotto cartesiano dà luogo a tuple con tutti gli attributi propri delle due relazioni. È possibile poi effettuare una selezione delle sole tuple logicamente collegate dalle due relazioni specificando un'appropriata condizione di selezione. Dato che questa sequenza costituita dal prodotto cartesiano seguito da una selezione è usata piuttosto frequentemente per identificare e selezionare tuple collegate di due relazioni, è stata ideata un'operazione speciale, detta Join, per specificarla con una singola operazione.

7.4.5 L'operazione di join

L'operazione di join, indicata con ⋈, è usata per unire tuple collegate, provenienti da due relazioni, in tuple singole. Essa è molto importante per ogni BdD relazionale che abbia più di una relazione, perché consente di eseguire associazioni tra relazioni.
La forma di un'operazione di join su due relazioni $R(A_1, A_2, \ldots, A_n)$ e $S(B_1, B_2, \ldots, B_m)$ è:

$$R \bowtie_{\text{<condizione di join>}} S$$

Il risultato del join è una relazione Q con n+m attributi $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in quest0ordine; Q presenta una tupla in corrispondenza a una combinazione di tuple – una di R e una di S – quando la combinazione soddisfa la condizione di join. Questa è a differenza principale tra il prodotto cartesiano e il join: in quest'ultimo compaiono solo le condizioni di tuple che soddisfano la condizione di join, mentre nel prodotto cartesiano sono incluse nel risultato tutte le combinazioni di tuple. La condizione di join è precisata sugli attributi delle due relazioni R e S ed è valutata per ogni combinazione di tuple. Ogni combinazione di tuple per la quale la condizione di join ha un valore vero è inserita nella relazione risultante Q come una singola tupla risultante dall'unione.
Una condizione di join generale assume la forma

<condizione>AND<condizione>AND…AND<condizione>

In cui ogni condizione assume la forma $A_i\ \theta\ B_j$; A_i è un attributo di R, B_j è un attributo di S e θ è uno degli operatori di confronto. Un operazione di join con una condizione di join così generale è detta theta join. Le tuple i cui attributi di join sono nulli non compaiono nel risultato. In questo senso l'operazione di join non conserva necessariamente tutte le informazioni presenti nelle relazioni dei partecipanti.
Il join più comune prevede condizioni di join con solo confronti di uguaglianza. Un join di questo tipo, in cui cioè il solo operatore di confronto usato è '=', è detto equijion. Si noti che nel risultato di un equijoin si hanno sempre una o più coppie di attributi con valori identici in ogni tupla.
Dato che, in ogni coppia di attributi con valori uguali, uno degli attributi è superfluo, è stata ideata una nuova operazione, detta join naturae – e indicata con * – per sbarazzarsi del secondo (superfluo) attributo presente in una condizione di equijoin. La definizione standard di join naturale richiede che i due attributi di join (od ogni coppia di attributi di join) abbiano lo stesso nome in entrambe le relazioni.
Una definizione generale per il join naturale è:

$$Q \leftarrow R *_{(\text{<lista1>}),(\text{<lista2>})} S$$

In questo caso <lista1> specifica un elenco di attributi in R e <lista2> in S. Le liste sono usate per formare condizioni di confronto di uguaglianza tra coppie di attributi corrispondenti; le condizioni vengono poi unite tramite AND. Nel risultato Q di tiene solo la lista corrispondente ad attributi della prima relazione R.
Se nessuna combinazione di tuple soddisfa la condizione di join, il risultato è una relazione vuota, senza cioè nessuna tupla.

7.4.6 Un insieme completo di operazioni dell'algebra relazionale.

Si può dimostrare che l'insieme delle operazioni dell'algebra relazionale {σ, π, ∪, –, X} è un insieme completo, cioè per ogni altra operazione dell'algebra relazionale può essere espressa come una sequenza di operazioni di questo insieme. Ad esempio intersezione:

$$R \cap S = (R \cup S) - ((R - S) \cup (S - R))$$

o il join

$$R \bowtie_{\text{<condizione>}} S = \sigma_{\text{<condizione>}} (R \times S)$$

7.4.7 L'operazione di divisione

L'operazione di divisione si applica a due relazioni $R(Z) \div S(X)$, dove $X \subseteq Z$. Sia $Y = Z - X$ (e, quindi $Z = X \cup Y$); sia, cioè, Y l'insieme di attributi di R che non sono attributi di S, Il risulato della divisione è una relazione T(Y) che comprende una tupla t se in R sono presenti tuple con $t_R[Y]=t$, e

con $t_R[X] = t_s$ per ogni tupla t_s di S. Ciò significa che, perché una tupla t compaia nel risultato T della divisione, in R devono comparire i valori di T in combinazione con ogni tupla di S.
L'operatore di divisione può essere espresso come una sequenza di operazioni π, X, – nel modo seguente:
$T_1 \leftarrow \pi_y(R)$
$T_2 \leftarrow \pi_y((S \times T_1) - R)$
$T \leftarrow T_1 - T_2$

R	A	B
	a1	b1
	a2	b1
	a3	b1
	a4	b1
	a1	b2
	a3	b2
	a2	b3
	a3	b3
	a4	b3
	a1	b4
	a2	b4
	a3	b4

S	A
	a1
	a2
	a3

T	B
	b1
	b4

$T \leftarrow R \div S$

7.5 Altre operazioni relazionali

7.5.1 Funzioni aggregate e raggruppamento

Il primo tipo di richiesta che non può essere espresso dall'algebra relazionale di base consiste nello specificare funzioni aggregate matematiche su collezioni di valori di BdD. Esempi di queste funzioni prevedono il recupero dello stipendio medio o totale di tutti gli impiegati, o il recupero del numero delle tuple impiegato. Funzioni comuni applicate a collezioni di valori numerici sono SUM, AVERAGE, MAXIMUM e MINIMUM, La funzione COUNT viene utilizzata per contare tuple o valori.
Un altro tipo comune di richiesta prevede il raggruppamento delle tuple presenti in una relazione sulla base del valore di alcuni loro attributi, e quindi l'applicazione di una funzione aggregata indipendente ad ogni gruppo.
La funzione aggregata viene così definita:

$$<\text{attributi di raggruppamento}>\mathcal{F}_{<\text{lista di funzioni}>}(R)$$

dove <attributi di raggruppamento> è una lista di attributi della relazione specificata in R, e <lista di funzioni> è una lista di coppie <funzione><attributo>. In ogni coppia di questi tipo, funzione è SUM o AVERAGE o MAXIMOM, MINIMUM o count e <attributo> è un attributo della relazione specificata R. La relazione risultante presenta gli attributi di raggruppamento più un attributo per ogni elemento della lista di funzioni.
Se non venissero specificati attributi di raggruppamento, le funzioni verrebbero applicate ai valori degli attributi di tutte le tuple della relazione, cosicché la relazione risultante avrebbe una sola tupla. I duplicati non sono eliminati.

7.5.2 Operazioni di chiusura ricorsiva

Un altro tipo di operazione che in generale non può essere specificata nell'algebra relazionale di base è la chiusura ricorsiva. Questa operazione è applicata ad un'associazione ricorsiva tra tuple dello stesso tipo, come ad esempio l'associazione tra un impiegato e un supervisore. Tale

associazione è descritto dalla chiave esterna SUPERSSN della relazione IMPIEGATO, che collega ogni tupla impiegato (nel ruolo di chi è sorvegliato) a un'altra tupla impiegato (nel ruolo di supervisore). Un esempio di operazione ricorsiva consiste nel recuperare tutti gli impiegato sorvegliati da un impiegato 'e' a tutti i livelli. Anche se in algebra relazionale è semplice specificare tutti gli impiegati sorvegliati ad uno specifico livello, è difficile specificare tutti quelli sorvegliati a tutti i livelli. Ad esempio, per specificare gli SSN di tutti gli impiegati sorvegliati direttamente dall'impiegato il cui nome è 'Marilyn Manson', si possono eseguire le seguenti operazioni:
MANSON _SSN ← $\pi_{SSN}(\sigma_{NOME_BATT='Marilyn' \text{ AND } COGNOME='Manson'}(IMPIEGATO))$
SUPERVISIONE(SSN1, SSN2) ← $\pi_{SSN, SUPERSSN}(IMPIEGATO)$
RISULTATO1(SSN) ← $\pi_{SSN1}(SUPERVISIONE \bowtie_{SSN2=SSN} MANSON_SSN)$
Per recuperare quelli al livello 2:
RISULTATO2(SSN) ← $\pi_{SSN1}(SUPERVISIONE \bowtie_{SSN2=SSN} RISULTATO1)$

7.5.3 Operazioni di join esterno e di unione esterna

Le operazioni di join descritte finora accoppiano tuple che soddisfano la condizione di join. Ad esmpio, per un'operazione di join naturale R*S, sono presenti solo le tuple di R che hanno tuple corrispondenti in S. Pertanto, tuple prive di una tupla corrispondente (matching) o collegata (related) vengono eliminate dal risultato del join. Vengono anche eliminate le tuple con valori nulli negli attributi di join. Quando si desidera mantenere nel risultato del join tutte le tuple di R, o quelle di S, o quelle presenti in entrambe le relazioni, abbiano esse o meno tuple corrispondenti nell'altra relazione, si può usare un insieme di operazioni, dette join esterni (outer joins). Ciò soddisfa la necessità di interrogazioni in cui le tuple provenienti da due tabelle devono essere combinate accoppiando righe corrispondenti, ma alcune tuple possono essere perse per mancanza di valori di accoppiamento. In questi casi sarebbe auspicabile poter disporre di un'operazione in grado di conservare tutte le tuple, che esse diano luogo o meno ad un'accoppiamento.
Ad esempio, si supponga di volere un elenco dei nomi di tutti gli impiegati, nonché il nome dei dipartimenti che essi dirigono ammesso che dirigano un dipartimento; per recuperare il risultato è possibile eseguire un'operazione di join estrerno sinistro, indicato con ⟕, nel modo seguente:
TEMP ← (IMPIEGATO $⟕_{SSN=SSN_DIR}$ DIPARTIMENTO)
RISULTATO ← $\pi_{NOME_BATT, INIZ, COGNOME, NOME_D}(TEMP)$
L'operazione di join esterno sinistro mantiene nel risultato ogni tupla presente nella prima relazione R di R ⟕ S, ossia la relazione di sinistra; se in S non si trova nessuna tupla di accoppiamento, allora gli attributi di S nel risultato del join sono completato o imbottiti con valori nulli.
Un'operazione simile è il join esterno destro ⟖e il join esterno completo.
L'operazione di unione esterna (outer union) è stata sviluppata per conservare l'unione di tuple provenienti da due relazioni se le relazioni non sono compatibili all'unione, ma basta che siano parzialmente compatibili (alcuni loro attributi sono compatibili). Ad esempio un'unione esterna può essere eseguita su due relazioni i cui schemi sono STUDENTE(Nome, SSN, Dipartimento, Tutor) e CORPO_DOCENTE(Nome, SSN, Dipartimento, Fascia). Lo schema di relazione risultante è R(Nome, SSN, Dipartimento, Tutor, Fascia), e nel risultato vengono inserite tutte le tuple provenienti da entrambe le relazioni; le tuple provenienti da studente avranno valore nullo per fascia, quelle da corpo_docente avranno valori nulli per Tutor.

8.1 La definizione dei dati e le modifiche agli schemi in SQL2

SQL utilizza i termini tabella, riga e colonna rispettivamente come sinonimi di relazione, tupla e attributo.

8.1.1 I concetti di schema e di catalogo

Uno schema SQL è identificato da un nome di schema e contiene un identificatore di autorizzazione per indicare l'utente o l'account che ne è proprietario, nonché i descrittori per ciascun elemento

nello schema. I suoi elementi comprendono le tabelle, i vincoli, le viste, i domini e altri costrutti che lo descrivono. Uno schema è creato attraverso l'istruzione create schema, che può includere le definizioni di tutti gli elementi dello schema. In alternativa, è possibile limitarsi ad assegnare un nome ed un identificatore di autorizzazione allo schema e definire gli elementi successivamente, Ad esempio, l'istruzione che seguen crea uno schema chiamato azienda, il cui proprietario è l'utente con l'identificatore di autorizzazione MMANSON:

CREATE SCHEMA AZIENDA AUTHORIZATION MMANSON

Oltre al concetto di schema in SQL2 esiste anche il concetto di catalogo, che consiste in una raccolta di schemi cui viene associate un nome in un ambiente SQL.

8.1.2 Il comando create table e i vincoli di tipi di dati.

Il comando create table è usato per specificare una nuova relazione assegnandole un nome e specificando i suoi attributi e vincoli. Gli attributi sono specificati per primi e a ciascun attributo è assegnato un nome, un tipo di dati per specificare il suo dominio di valori e gli eventuali vincoli sui valori come, ad esempio, not null. La chiave, vincolo di integrità dell'entità, e i vincoli d'integrità referenziale possono essere specificati direttamente all'interno dell'istruzione create table, dopo che sono stati dichiarati gli attributi; in alternativa li si può aggiungere successivamente usando il comando alter table.
Di solito lo schema in cui vengono dichiarate le relazioni è specificato implicitamente dall'ambiente in cui vengono eseguite le istruzioni create table, ma lo si può anche indicare esplicitamente:

CREATE TABLE IMPIEGATO

o

CREATE TABLE AZIENDA.IMPIEGATO

Domini e tipi di dati in SQL2: i tipi di dati disponibili sono:

- Tipi di dati numerici, tra cui INT, SMALLINT, FLOAT, REAL, DOUBLE PRECISION…
- Stringhe di caratteri: di lunghezza fissa come CHAR(n) o CHARACTER(n), o di lunghezza variabile: VARCHAR(n), in cui n è la lunghezza massima.
- Stringhe di bit: BIT(n) o BIT VARYING(n)
- Data e ora: DATE ha 10 posizioni e i suoi componenti sono YEAR, MONTH e DAY, tipicamente nella forma YYYY-MM-DD. Il tipo di adti TIME ha otto posizioni (HOUR, MINUTE, SECOND) nella forma HH:MM:SS

In SQL2 è possibile specificare direttamente il tipo di dati di ogni attributo, alternativamente può essere dichiarato il dominio e il nome del dominio usato:

CREATE DOMAIN SSN_TYPE AS CHAR(9)

Specificazione dei vincoli e dei valori predefiniti in SQL2. Poiché SQL accetta NULL come valore degli attributi, può essere specificato il vincolo NOT NULL. È possibile anche specificare un valore predefinito per un attributo aggiungendo la clausola DEFAULT <value>. Se non è specificata alcuna clausola predefinita, il valore predefinito di default è null.
Dopo aver specificato i vincoli sugli attributi (colonne), è possibile specificare vincoli di tabella, tra cui quelli di chiave e di integrità referenziale. La clausola PRIMARY KEY specifica uno o più attributi che costituiscono la chiave primaria di una relazione. La clausola UNIQUE specifica le chiavi ulteriori (o secondarie). L'integrità referenziale è specificata attraverso la clausola FOREIGN KEY.
Un vincolo di integrità referenziale può essere violato in caso di cancellazione di tuple o modifica… per questo si può specificare l'azione referenziale innescata. Le opzioni includono SET NULL, CASCADE, SET DEFAULT e devono essere qualificate con ON DELETE oppure con ON UPDATE.
Ad un vincolo può anche essere associato un nome, con la parola chiave CONSTRAINT. I nomi di tutti i vincoli all'interno di un particolare schema devono essere unici.

8.1.3 I comandi DROP SCHEMA e DROP TABLE.

Se un intero schema non è più necessario, può essere utilizzato il comando drop schema. Esistono due opzioni relative al comportamento di eliminazione: CASCADE e RESTRICT. Ad esempio, per eliminare lo schema della BdD AZIENDA e tutte le sue tabelle, i domini e gli altri elementi, l'opzione CASCADE va usata nel modo seguente:

DROP SCHEMA AZIENDA CASCADE

Se viene scelta l'opzione restrict lo schema viene eliminato solo se non contiene elementi.
Se una relazione non è più necessaria, si usa il comando DROP TABLE. In questo caso l'opzione RESTRICT serve ad indicare di cancellare la relazione solo se non viene fatto alcun riferimento ad essa in un qualsiasi vincolo o vista; con l'opzione CASCADE vengono cancellati anche i vincoli e le viste.

8.1.4 Il comando ALTER TABLE

La definizione di una tabella base può essere cambiate usando il camndo alter table, che è un comando di evoluzione dello schema.Le possibili operazioni di modifica delle tabelle comprendono l'aggiunta o l'eliminazione di una colonna, la modifica di una definizione di colonna e l'aggiunta o l'eliminazione di vincoli di tabella. Ad esempio, per inserire un attributo:

ALTER TABLE AZIENDA.IMPIEGATO ADD LAVORO VARCHAR(12)

O per eliminare l'attributo indirizzo dalla tabella base IMPIEGATO:

ALTER TABLE AZIENDA.IMPIEGATO DROP INDIRIZZO CASCADE

O per cambiare i valori di default:

ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN DROP DEFAULT;
ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN SET DEFAULT "333445555";

O per inserire o eliminare nuovi vincoli:

ALTER TABLE COMPANY.EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;

8.2 Interrogazioni fondamentali in SQL2

SQL ha un'istruzione fondamentale per recuperare le informazioni da una BdD: l'istruzione SELECT. SQL consente ad una tabella (relazione) di avere due più tuple identiche in tutti i valori dei loro attributi. In generale una tabella SQL non è un insieme di tuple, perché un insieme non permette di avere membri uguali; piuttosto è un multinsieme, talvolta chiamata sacca (bag) di tuple. Alcune relazioni SQL non possono che essere insiemi perché è stato dichiarato un vincolo di chiave o perché l'operazione DISTINCT è stata usata con l'istruzione SELECT.

8.2.1 La struttura SELECT – FROM – WHERE delle interrogazioni SQL

La forma fondamentale dell'istruzione select, talvolta chiamata blocco select-from-where, è formata dalle tre clausole SELECT, FROM e WHERE e ha la seguente forma:
SELECT <elenco di attributi>
FROM <elenco tabelle>
WHERE <condizione>
dove:
<elenco di attributi> è una lista di attributi i cui valori devono essere recuperati
<elenco tabelle> è un elenco dei nomi delle relazioni necessarie per eseguire l'interrogazione
<condizione> è un'espressione booleana che identifica le tuple che devono essere recuperate dall'interrogazione

INTERROGAZIONE 0
Trova la data di nascita del o degli impiegati il cui nome è 'John B. Smith'

Q0:
```
SELECT  BDATE, ADDRESS
 FROM  EMPLOYEE
 WHERE  FNAME='John' AND MINIT='B' AND LNAME='Smith';
```

Q0 è simile alla seguente espressione dell'algebra relazionale, eccetto per i duplicati che, se ci sono, non vengono eliminati.
πBDATE,ADDRESS (σFNAME='John' AND MINIT='B' AND LNAME='Smith' (EMPLOYEE))

INTERROGAZIONE 1
Trova il nome e l'indirizzo di tutti gli impiegati che lavorano per il dipartimento 'ricerca'

Q1:
```
SELECT  FNAME, LNAME, ADDRESS
 FROM  EMPLOYEE, DEPARTMENT
 WHERE  DNAME='Research' AND DNUMBER=DNO;
```

INTERROGAZIONE 2
Per tutti i progetti locati a 'Stafford', si elenchi il numero del progetto, il numero del dipartimento che lo controlla, nonché il cognome, l'indirizzo e la data di nascita del direttore del dipartimento.

Q2:
```
SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
 FROM  PROJECT, DEPARTMENT, EMPLOYEE
 WHERE  DNUM=DNUMBER AND MGRSSN=SSN AND PLOCATION='Stafford';
```

8.2.2 La problematica dei nomi di attributi ambigui e la loro ridenominazione (assegnando pseudonimi)

Nel SQL lo stesso nome può venir usato per due o più attributi a patto che essi siano in relazioni differenti. Se succede, bisogna qualificare il nome dell'attributo per evitare ambiguità:

Q1A:
```
SELECT FNAME, EMPLOYEE.NAME, ADDRESS
 FROM EMPLOYEE, DEPARTMENT
 WHERE  DEPARTMENT.NAME='Research' AND
DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER;
```

Ambiguità possono sorgere anche nel caso di interrogazioni che si riferiscono due volte alla medesima relazione, come nel caso seguente:

INTERROGAZIONE 8
Per ogni impiegato, si recuperi il suo nome, cognome e il nome e il cognome del suo immediato supervisore

Q8:
```
SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
 FROM  EMPLOYEE AS E, EMPLOYEE AS S            ← aliasing
 WHERE  E.SUPERSSN=S.SSN;
```

È possibile anche rinominare tutti gli attributi:

EMPLOYEE AS E(FN, MI, LN, SSN, BD, ADDR, SEX, SAL, SSSN, DNO)

8.2.3 La clausola WHERE non specificata e l'uso dell'asterisco (*)

Se in una select manca la clausola where, questa indica che non viene usata alcuna condizione per la selezione delle tuple; quindi tutte le tuple delle relazione specificata nella clausola FROM sono considerate valide e costituiscono il risultato dell'interrogazione. Se nella clausola FROM più di una relazione è specificata e non vi è una clausola where, viene eseguito il prodotto cartesiano, ottenendo tutte le possibili combinazioni di tuple di queste relazioni.

Interrogazioni 9 and 10
Seleziona tutti gli SSN degli impiegati (Q9) e tutte le combinazioni di EMPLYEE SSN e nome del dipartimento della base di dati

```
Q9:
SELECT  SSN
FROM  EMPLOYEE;

Q10:
SELECT  SSN, DNAME
 FROM  EMPLOYEE, DEPARTMENT;
```

Per recuperare invece tutti i valori degli attributi delle tuple selezionati, non si devono elencare esplicitamente tutti gli attributi, basta specificare un asterisco (*) che sta per tutti gli attributi:
Per esempio l'interrogazione Q1C recupera i valori di tutte le tuple degli impiegati che lavorano per il dipartimento 5; l'interrogazione Q1D recupera tutti gli attributi di impiegato e tutti gli attributi di dipartimento in cui il dipendente lavora per ogni dipendente del dipartimento 'Research'; Q10 specifica il prodotto cartesiano delle relazioni impiegato e dipartimento

```
Q1C:
SELECT  *
 FROM  EMPLOYEE
 WHERE  DNO=5;

Q1D:
SELECT  *
 FROM  EMPLOYEE, DEPARTMENT
 WHERE  DNAME='Research' AND DNO=DNUMBER;

Q10A:
SELECT  *
 FROM  EMPLOYEE, DEPARTMENT;
```

8.2.4 Le tabelle come insiemi in SQL

SQL di solito considera una tabella non come un insieme, ma come un multinsieme; tuple con identici valori degli attributi possono comparire più di una volta in una tabella e nel risultato di un'interrogazione. SQL non elimina automaticamente le tuple duplicate nei risultati delle interrogazioni per i seguenti motivi:

- L'eliminazione dei duplicati è un'operazione costosa (un modo per implementarla è ordinare prima le tuple e poi eliminare i duplicati);
- L'utente può volere le tuple duplicate nel risultato di un'interrogazione
- Quando una funzione di aggregazione è applicata alle tuple, nella maggior parte dei casi non si vogliono eliminare i duplicati.

Quando si desidera eliminare le tuple duplicate dal risultato di un'interrogazione SQL, si utilizza la parola chiave distinct nella clausola select, il che significa ce solo le tuple distinte dovrebbero rimanere nel risultato, In generale un'interrogazione con select distinct elimina i duplicati mentre un'interrogazione con select all non lo fa ("all" è il valore di default).

INTERROGAZIONE 11
Trova il salario di ogni impiegato (Q11) e tutti i valori distinti di salario (Q11A).

```
Q11:
SELECT ALL  SALARY
 FROM  EMPLOYEE;

Q11A:
SELECT DISTINCT  SALARY
 FROM  EMPLOYEE;
```

Alcune operazioni insiemistiche dell'algebra relazionale sono state incorporate direttamente in SQL. Vi è l'operazione in unione insiemistica (Union), differenza (Except) e intersezione (intersect). Le relazioni derivanti da esse sono insiemi di tuple, e cioè le tuple duplicate sono eliminate dal risultato. Queste operazioni si applicano solo a relazioni compatibili all'unione.

INTERROGAZIONE 4
Fai una lista di tutti i numeri di progetti a cui lavorano gli impiegati che di cognome fanno 'Smith', sia come impiegato sia come direttore del dipartimento che controlla il progetto.

```
Q4:
(SELECT  DISTINCT PNUMBER
FROM  PROJECT, DEPARTMENT, EMPLOYEE
 WHERE  DNUM=DNUMBER AND MGRSSN=SSN AND LNAME='Smith')
 UNION
 (SELECT  DISTINCT PNUMBER
 FROM  PROJECT, WORKS_ON, EMPLOYEE
 WHERE  PNUMBER=PNO AND ESSN=SSN AND LNAME='Smith');
```

8.2.5 Confronti di sottostringhe, operatori aritmetici e ordinamento

La prima funzione permette di eseguire un confronto solo su parti di una stringa di caratteri, usando l'operatore di confronto LIKE. Le stringhe parziali sono specificate utilizzando due caratteri riservati: il segno percentuale % sostituisce un numero arbitrario di caratteri e il segno di sottolineatura _ sostituisce un solo carattere.

INTERROGAZIONE 12
Trova tutti gli impiegati che abitano a Houston, Texas.

```
Q12:
SELECT  FNAME, LNAME
```

```
 FROM  EMPLOYEE
 WHERE  ADDRESS LIKE '%Houston,TX%';
```

INTERROGAZIONE 12A
Trova tutti gli impiegati nati negli anni 50

```
 Q12A:
 SELECT  FNAME, LNAME
  FROM  EMPLOYEE
  WHERE  BDATE LIKE'_ _ 5 _ _ _ _ _ _ _';
```

Altre funzionalità comprendono l'uso di operatori aritmetici standard: addizione (+), sottrazione (-), moltiplicazione (*), e divisione (/) che possono essere applicati a valori numerici o attributi con domini numerici.

INTERROGAZIONE 13
Mostra i salari che risulterebbero aumentando del 10% i salari degli impiegati che lavorano al progetto 'ProductX'.

```
 Q13:
 SELECT  FNAME, LNAME, 1.1*SALARY
  FROM  EMPLOYEE, WORKS_ON, PROJECT
  WHERE  SSN=ESSN AND PNO=PNUMBER AND PNAME='ProductX';
```

Per le stringhe c'è l'operatore di concatenazione '| |' che può essere usato in un interrogazione per giustapporre e due valori stringa. Per i tipi di dati data, ora, timestamp e intervallo ci sono gli operatori aumento (+) o diminuzione (-) di una data, ora, timestamp o intervallo di una quantità compatibile col tipo di dati.
Inoltre si può specificare un intervallo come differenza tra due valori data, ora, timestamp. Un altro operatore di confronto che può essere usato per comodità è BETWEEN:

INTERROGAZIONE 14
Trova tutti gli impiegati che lavorano al dipartimento 5 il cui salario è compreso tra $30,000 e $40,000.

```
 Q14:
 SELECT  *
  FROM  EMPLOYEE
  WHERE  (SALARY BETWEEN 30000 AND 40000) AND DNO = 5;
```

SQL permette agli utenti di ordinare le tuple risultanti da un'interrogazione a seconda di uno o più attributi, utilizzando la clausola ORDER-BY:

INTERROGAZIONE 15
Trova la lista di tutti gli impiegati e i progetti su cui stanno lavorando, ordinati per dipartimento e, all'interno di ogni dipartimento, ordinati alfabeticamente per cognome e nome.

```
 Q15:
 SELECT  DNAME, LNAME, FNAME, PNAME
  FROM  DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
```

```
WHERE  DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER
ORDER BY  DNAME, LNAME, FNAME;
```

L'ordine di default è A-Z. Possiamo specificare con la parola chiave DESC se vogliamo l'ordine inverso:

```
ORDER BY  DNAME DESC, LNAME ASC, FNAME ASC
```

8.3 Interrogazioni SQL più complesse

8.3.1 Interrogazioni nidificate e confronti di insiemi

Alcune interrogazioni richiedono di estrarre dei valori esistenti della BdD per poi usarli in una condizione di confronto. Tali interrogazioni possono essere semplicemente formulate usando delle interrogazioni nidificate, che sono blocchi completi di SELECT . . . FROM . . . WHERE . . . posti all'interno della clausola where di un'altra interrogazione chiamata interrogazione esterna.

```
Q4A:
SELECT  DISTINCT PNUMBER
 FROM  PROJECT
 WHERE  PNUMBER IN  ( SELECT  PNUMBER
                      FROM  PROJECT, DEPARTMENT, EMPLOYEE
                       WHERE  DNUM=DNUMBER AND MGRSSN=SSN AND
                       LNAME='Smith')
 OR
 PNUMBER IN  (     SELECT  PNO
                   FROM  WORKS_ON, EMPLOYEE
                   WHERE  ESSN=SSN AND LNAME='Smith');
```

La prima interrogazione nidificata selezione i numeri di progetto del dipartimento il cui dirigente è Smith, la seconda seleziona progetti in cui lavora Smith.
Nell'interrogazione esterna si seleziona una tupla progetto se il valore PNUMBER di quella tupla fa aprte di una delle due interrogazioni nidificate. L'operatore di confronto IN confronta un valore v con un insieme di valori V e restituisce TRUE se v è uno degli elementi di V.
L'operatore IN può anche confrontare una tupla di valori tra parentesi con un insieme di tuple compatibili all'unione. Ad esempio:

```
SELECT  DISTINCT ESSN
 FROM  WORKS_ON
 WHERE  (PNO, HOURS) IN  (SELECT PNO, HOURS FROM WORKS_ON WHERE
                          SSN='123456789');
```

Selezionerà i numeri di servizio sanitario di tutti i dipendenti che hanno la stessa combinazione (PNO, HOURS) su un progetto in cui lavora il dipendente John Smith (il cui SSN è 123456789).
Oltre all'operatore IN, possono essere usati altri operatori di confronto per confrontare un singolo valore v (tipicamente un nome di attributo) con un insieme o un multinsieme V (tipicamente il risultato di un'interrogazione nidificata). L'operatore = ANY restituisce TRUE se il valore v è uguale a un qualsiasi valore nell'insieme V ed è quindi equivalente a IN. Altri operatori che possono essere usati con ANY o ALL sono > >=, <, <= e <>. Ad esempio la condizione di confronto (v > ALL V) restituisce TRUE se il valore v è maggiore di tutti i valori dell'insieme V.

Un esempio è l'interrogazione seguente, che restituisce i nomi dei dipendenti il cui stipendi è maggiore di quello di tutti i dipendenti del dipartimento 5:

```
SELECT  LNAME, FNAME
 FROM  EMPLOYEE
 WHERE  SALARY > ALL (SELECT SALARY FROM EMPLOYEE WHERE DNO=5);
```

In caso di ambiguità riferimenti ad attributi non qualificati si riferiscono alla relazione dichiarata nell'interrogazione nidificata più interna:

INTERROGAZIONE 16
Trova il nome degli impiegati che hanno una persona a carico con lo stesso nome e lo stesso sesso dell'impiegato.

```
Q16:
SELECT  E.FNAME, E.LNAME
FROM  EMPLOYEE AS E
WHERE  E.SSN IN (   SELECT  ESSN
                    FROM  DEPENDENT
                    WHERE  E.FNAME=DEPENDENT_NAME AND E.SEX=SEX);
```

Le interrogazioni nidificate correlate. Quando una condizione nella clausola WHERE di una interrogazione nidificata si riferisce ad attributi di una relazione dichiarata nell'interrogazione esterna, si dice che le due interrogazioni sono correlate. Si può capire meglio il coneto di interrogazione correlata considerando che l'interrogazione nidificata è valutata una volta per ciascuna tupla (o combinazione di tuple) dell'interrogazione esterna.

```
Q16A:
SELECT  E.FNAME, E.LNAME
 FROM  EMPLOYEE AS E, DEPENDENT AS D
 WHERE  E.SSN=D.ESSN AND E.SEX=D.SEX AND E.FNAME=D.DEPENDENT_NAME;
```

L'implementazione originale dell'SQL conteneva anche l'operatore di confronto CONTAINS, che è usato per confrontare sue insiemi. L'operatore CONTAINS compara due insiemi di valori e ritorna TRUE se uno contiene tutti i valori dell'altro.

INTERROGAZIONE 3
Ritrova I nomi di tutti gli impiegato che lavorano su progetti controllati dal dipartimento 5.

```
Q3:
SELECT  FNAME, LNAME
 FROM  EMPLOYEE
 WHERE  ( (SELECT  PNO
                FROM  WORKS_ON
                WHERE  SSN=ESSN)
                CONTAINS
                 (SELECT  PNUMBER
                FROM  PROJECT
                WHERE  DNUM=5) );
```

8.3.2 Le funzioni EXISTS e UNIQUE in SQL

La funzione EXISTS in SQL è usata per controllare se il risultato di un'interrogazione nidificata e correlata è vuoto (non contiene tuple) oppure no:

```
Q16B:
SELECT  E.FNAME, E.LNAME
 FROM  EMPLOYEE AS E
 WHERE  EXISTS  ( SELECT  *
                  FROM  DEPENDENT
                  WHERE  E.SSN=ESSN AND E.SEX=SEX AND
                  E.FNAME=DEPENDENT_NAME);
```

Le funzioni EXISTS e NOT EXISTS sono generalmente usate insieme ad interrogazioni nidificate o correlate. NOT EXISTS ritorna TRUE se non vi sono tuple nel risultato di un'interrogazione:

INTERROGAZIONE 6
Ritrova i nomi dei dipendenti che non hanno persone a carico.

```
Q6:
SELECT  FNAME, LNAME
 FROM  EMPLOYEE
 WHERE  NOT EXISTS  (SELECT  *
                     FROM  DEPENDENT
                     WHERE  SSN=ESSN);
```

INTERROGAZIONE 7
Si elenchino i nomi dei manager che hanno almeno un dipendente

```
Q7:
SELECT  FNAME, LNAME
 FROM  EMPLOYEE
 WHERE  EXISTS  (SELECT  *
                 FROM  DEPENDENT
                 WHERE  SSN=ESSN)
        AND
        EXISTS
                 (SELECT  *
                  FROM  DEPARTMENT
                  WHERE  SSN=MGRSSN);
```

INTERROGAZIONE 3, che era stata usata per illustrare l'operatore CONTAINS può venire riscritta usando EXISTS e NOT EXISTS, perché (S1 CONTAINS S2) è logicamente equivalente a dire se (S2 EXCEPT S1) è vuoto:

```
Q3A:
SELECT  FNAME, LNAME
 FROM  EMPLOYEE
 WHERE  NOT EXISTS
                          ((SELECT  PNUMBER
                           FROM  PROJECT
                           WHERE  DNUM=5)
                           EXCEPT
                           (SELECT  PNO
                           FROM  WORKS_ON
                           WHERE  SSN=ESSN));
```

8.3.3 Insiemi espliciti e NULL

È possibile usare anche un insieme esplicito di valori nella clausola where, piuttosto che un'interrogazione nidificate. Tale insieme va posto tra parentesi in SQL.

INTERROGAZIONE 17
Trova tutti i SSN degli impiegati che lavorano ai progetti 1, 2 o 3

```
Q17:
SELECT  DISTINCT ESSN
 FROM  WORKS_ON
 WHERE  PNO IN (1, 2, 3);
```

SQL constente interrogazioni per controllare se un valore è NULL – mancante, indefinito o non applicabile. Tuttavia, invece di usare l'= o comparare un attributo a NULL, SQL usa IS o IS NOT:

INTERROGAZIONE 18
Trova il nome di tutti gli impiegati che non hanno un supervisore

```
Q18:
SELECT  FNAME, LNAME
FROM  EMPLOYEE
WHERE  SUPERSSN IS NULL;
```

8.3.4 Ridenominazione degli attributi e delle tabelle collegate con i join

È possibile rinominare un attributo qualsiasi che appare nel risultato di un'interrogazione aggiungendo un qualificatore AS seguito dal nome desiderato. Il costrutto AS può essere usato, sia nella clausola SELECT sia nella clausola FROM per assegnare uno pseudonimo ai nomi sia degli attributi sia delle relazioni:

```
Q8A:
SELECT  E.LNAME AS EMPLOYEE_NAME, S.LNAME AS SUPERVISOR_NAME
 FROM  EMPLOYEE AS E, EMPLOYEE AS S
 WHERE  E.SUPERSSN=S.SSN;
```

Il concetto di una tabella collegata via join (o relazione collegata via join) è stato incorporato in SQL2 per permettere agli utenti di specificare una tabella risultante da un'operazione di join

specificata nella clausola FROM di un'interrogazione. Questo costrutto può essere più facile da capire rispetto al mescolare insieme tutte le frasi di selezione e le condizioni di join nella clausola where:

```
Q1A:
SELECT  FNAME, LNAME, ADDRESS
 FROM  (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
 WHERE  DNAME='Research';
```

Il concetto di tabella collegata via join consente anche all'utente di specificare differenti tipi di join, ad esempio il natural join e I vari tipi di outer join. In un natural join su due relazioni R e S, non è specificata alcuna condizione di join; viene creata una condizione implicita di equijoin per ciascuna coppia di attributi con lo stesso nome presenti in R e S. Ogni coppia di attributi di questo tipo è inclusa una sola volta nella relazione risultante.
Se i nomi degli attributi di join non sono gli stessi delle relazioni base, è possibile rinominare gli attributi in modo che essi corrispondano e poi applicare il natural join. In questo caso il costrutto AS può essere usato per cambiare il nome a una relazione e a tutti i suoi attributi nella clausola from:

```
Q1B:
SELECT  FNAME, LNAME, ADDRESS
FROM  (EMPLOYEE NATURAL JOIN (DEPARTMENT AS DEPT (DNAME, DNO, MSSN,
MSDATE)))
WHERE  DNAME='Research;
```

Il tipo predefinito di join in una tabella collegata con i join è un join interno, in cui una tupla è contenuta nel risultato se e solo se esiste una tupla corrispondete nell'altra relazione.
Esistono però anche gli OUTER JOIN:

```
Q8B:
SELECT  E.LNAME AS EMPLOYEE_NAME, S.LNAME AS SUPERVISOR_NAME
 FROM  (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S ON E.SUPERSSN=S.SSN);
```

Le opzioni disponibili per specificare le tabelle collegate via join in SQL2 comprendono INNER JOIN (uguale a JOIN), LEFT OUTER JOIN, RIGHT OUTER JOIN e FULL OUTER JOIN. Negli ultimi tre la parola chiave OUTER può essere omessa. È possibile anche nidificare le specificazioni via join, cioè una delle tabelle operando in un join può essere essa stessa una tabella collegata via join:

```
Q2A:
SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
FROM  ((PROJECT JOIN DEPARTMENT ON DNUM= DNUMBER) JOIN EMPLOYEE ON
MGRSSN=SSN)
WHERE  PLOCATION='Stafford';
```

8.3.5 Funzioni di aggregazione e raggruppamento

Poichè le funzioni di aggregazione e di raggruppamento sono richieste in molte applicazioni di BdD, SQL comprende: COUNT, SUM, MAX, MIN, e AVG.
La funzione COUNT restituisce il numero di tuple o valori individuati da un'interrogazione. Le funzioni SUM, MAX, MIN e AVG sono applicate a un insieme o a un multiinsieme di valori numerici e restituiscono rispettivamente la somma, il massimo, il minimo o la media di quei valori.

Queste funzioni possono essere usate nella clausola select o nella clausola having. Le funzioni MAX e MIN possono essere utilizzate anche con attributi che hano domini non numerici se tra i valori del dominio è definito un ordinamento totale.

INTERROGAZIONE 19
Trova la somma dei salari di tutti gli impiegati, il salario Massimo, minimo e medio.

```
Q19:
SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY), AVG (SALARY)
FROM  EMPLOYEE;
```

INTERROGAZIONE 20
Trova la soma degli stipendi di tutti gli impiegati del dipartimento di ricerca, e il salario Massimo, minimo e medio di quell diparimento.

```
Q20:
SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY), AVG (SALARY)
FROM  EMPLOYEE, DEPARTMENT
WHERE  DNO=DNUMBER AND DNAME='Research';
```

INTERROGAZIONI 21 and 22
Trova il numero totale di impiegati della compagnia (Q21) e il numero di impiegati del dipartimento di ricerca (Q22).

```
Q21:
SELECT  COUNT (*)
FROM  EMPLOYEE;
```

```
Q22:
SELECT  COUNT (*)
FROM  EMPLOYEE, DEPARTMENT
WHERE DNO=DNUMBER AND DNAME='Research';
```

INTERROGAZIONE 23
Conta il numero di salari distinti

```
Q23:
SELECT  COUNT (DISTINCT SALARY)
FROM EMPLOYEE;
```

In alcuni casi potremmo aver bisogno di applicare queste funzioni ad un insieme di tuple selezionate. Ad esempio, per trovare il nome di tutti gli impiegati che hanno 2 o più persone a carico:

```
Q5:
SELECT  LNAME, FNAME
FROM  EMPLOYEE
```

```
WHERE  (SELECT COUNT (*)
            FROM DEPENDENT
            WHERE  SSN=ESSN) >= 2;
```

In molti casi si vogliono applicare le funzioni di aggregazione per creare sottogruppi di tuple in una relazione, sulla base dei valori di un attributo. Ad esempio, si può voler trovare lo stipendio medio dei dipendenti di ciascun dipartimento o il numero di dipendenti che lavorano su ciascun progetto. In questi casi è necessario raggruppare le tuple che hanno lo stesso valore di uno o più attributi chiamati attributi di raggruppamento e applicare la funzione a ognuno di questi gruppi indipendentemente. SQL ha per questo scopo la clausola GROUP BY la quale specifica gli attributi di raggruppamento, che devono apparire anche nella clausola SELECT, in modo che il valore risultante dell'applicazione di una funzione a un gruppo di tuple compaia insieme al valore dell'attributo o degli attributi di raggruppamento.

INTERROGAZIONE 24
Per ogni dipartimento, ritrova il numero di dipartimento, il numeri di impiegati del dipartimento e il salario medio.

```
Q24:
SELECT  DNO, COUNT (*), AVG (SALARY)
FROM EMPLOYEE
GROUP BY  DNO;
```

INTERROGAZIONE 25
Per ogni progetto trova il numero, il nome e il numero degli impiegati che ci lavora sopra.

```
Q25:
SELECT PNUMBER, PNAME, COUNT (*)
FROM PROJECT, WORKS_ON
WHERE PNUMBER=PNO
GROUP BY PNUMBER, PNAME;
```

Talvolta si vogliono recuperare i valori di queste funzioni solo per gruppi che soddisfano certe condizioni. Ad esempio, si supponga di voler modificare l'interrogazione precedente in modo che solo i progetti con più di due impiegati appaiano nella nel risultato. SQL mette a disposizione una clausola HAVING, che può comparire insieme alla clausola GRUP BY. HAVING forcine una condizione sul gruppo di tuple associata al valore degli attributi di raggruppamento; solo i gruppi che soddisfano la condizione sono inclusi nel risultato dell'interrogazione.

INTERROGAZIONE 26
Per ogni progetto in cui lavorano più di due impiegati, trova il numero del progetto, il nome e il numero di impiegati che lavora su quel progetto.

```
Q26:
SELECT PNUMBER, PNAME, COUNT (*)
FROM PROJECT, WORKS_ON
```

```
WHERE PNUMBER=PNO
GROUP BY PNUMBER, PNAME
HAVING COUNT (*) > 2;
```

Si noti che mentre le condizioni di selezione nella clausola where limitano le tuple a cui vanno applicate le funzioni, la clausola having serve per selezionare interi gruppi.

INTERROGAZIONE 27
Per ogni progetto, ritroba ikl numero, il nome e il numero di impiegato del dipartimento 5 che ci lavora.

```
Q27:
SELECT  PNUMBER, PNAME, COUNT (*)
FROM  PROJECT, WORKS_ON, EMPLOYEE
WHERE  PNUMBER=PNO AND SSN=ESSN AND DNO=5
GROUP BY PNUMBER, PNAME;
```

Supponiamo che vogliamo trovare il numero degli impiegati il cui salario è maggiore di $40.000, ma solo per i dipartimenti che hanno più di cinque impiegati. La seguente interrogazione è scorretta:

```
SELECT DNAME, COUNT (*)
FROM  DEPARTMENT, EMPLOYEE
WHERE DNUMBER=DNO AND SALARY>40000
GROUP BY DNAME
HAVING COUNT (*) > 5;
```

È scorretta perché seleziona solo i dipartimenti che hanno più di cinque dipendenti che guadagnano più di $40.000. La regola è che la clausola where viene applicata prima, per selezionare le tuple singole; la clausola HAVING è applicata dopo per scegliere gruppi di tuple.. Le tuple, perciò sono già state limitate agli impiegati che guadagnano più di $40.000, prima che sia applicata la funzione HAVING. L'interrogazione corretta è la seguente:

INTERROGAZIONE 28
Per ogni dipartimento con più di cinque impiegati, ritrova il numero del dipartimento e il numero degli impiegati che guadagnano più di $40,000.

```
Q28:
SELECT  DNUMBER, COUNT (*)
FROM DEPARTMENT, EMPLOYEE
WHERE DNUMBER=DNO AND SALARY>40000 AND
                    DNO IN (SELECT DNO
                    FROM EMPLOYEE
                    GROUP BY  DNO
                    HAVING  COUNT (*) > 5)
GROUP BY  DNUMBER;
```

8.4 Le istruzioni Insert, Delete, and Update in SQL

8.4.1 Il comando INSERT

Nella sua forma più semplice insert viene usato per aggiungere una singola tupla in una relazione. Si devono specificare il nome della relazione e un elenco di valori per la tupla. I valori dovrebbero essere elencati nello stesso ordine in cui gli attributi corrispondenti non stati specificati dal comando create table:

U1:
INSERT INTO EMPLOYEE
VALUES (‘Richard’, ‘K’, ‘Marini’, ‘653298653’, ‘1962-12-30’,‘98 Oak Forest,Katy,TX’,‘M’, 37000, ‘987654321’, 4);

Una seconda forma dell’istruzione insert consente all’utente di specificare esplicitamente I nomi degli attributi che corrispondono ai valori forniti nel comando insert. Ciò è utile se una relazione ha molti attributi, ma solo ad alcuni di essi vanno assegnati i valori nella nuova tupla. Questi attributi dvono includere tutti quelli con la specificazione NOT NULL e nessun valore di default, ad esempio:

U1A:
INSERT INTO EMPLOYEE (FNAME, LNAME, DNO, SSN)
VALUES (‘Richard’, ‘Marini’, 4, ‘653298653’);

8.4.2 Il comando DELETE

Il comando delete rimuove tuple da una relazione. Incolude una clausola WHERE, simile a quelle delle interrogazioni SQL, per selezionare le tuple che devono venir cancellate.

U4A:
DELETE FROM EMPLOYEE
WHERE LNAME=‘Brown’;

U4B:
DELETE FROM EMPLOYEE
WHERE SSN=‘123456789’;

U4C:
DELETE FROM EMPLOYEE
WHERE DNO IN (SELECT DNUMBER
FROM DEPARTMENT
WHERE DNAME=‘Research’);

U4D:
DELETE FROM EMPLOYEE;

8.4.3 Il comando UPDATE

Il comando UPDATE è usato per modificare I valori degli attributi di una o più tuple selezionate. Come nel comando delete, con una clausola where si selezionano le tuple che devono venir modificate, mentre un’ulterioreclausola SET specifica gli attributi da modificare e i loro nuovi valori:

U5:
```
UPDATE PROJECT
SET PLOCATION = 'Bellaire', DNUM = 5
WHERE PNUMBER=10;
```

U6:
```
UPDATE EMPLOYEE
SET SALARY = SALARY *1.1
WHERE DNO IN (SELECT DNUMBER
                    FROM DEPARTMENT
                    WHERE DNAME='Research');
```

Si può anche specificare i valori NULL o DEFAULT come nuovo valore.

8.5 Viste (tabelle virtuali) in SQL

8.5.1 Il concetto di vista in SQL

Una vista nella terminologia SQL è una singola tabella che deriva da altre tabelle, che possono essere tabelle base o altre viste definite precedentemente. Una vista non esiste necessariamente: è considerata una tabella virtuale, in contrapposizione alle tabelle base le cui tuple sono effettivamente memorizzate nella base di dati. Ciò limita molto le operazioni di aggiornamento possibili che possono essere applicate ad essa, ma non pone alcuna limitazione alla sua interrogazione. Si può considerare una vista come un modo per specificare una tabella che si deve utilizzare di frequente anche se non può esistere fisicamente.

8.5.2 Specificazione delle viste

Il comando per specificare una vista è CREATE VIEW. Alla vista viene assegnato un nome di una tabella (virtuale) (o nome di vista), un elenco di nomi di attributi e un'interrogazione per specificare i contenuti della vista. Se nessuno degli attributi della vista è il risultato dell'applicazione di funzioni o di operazioni matematiche, non è necessario specificare i nomi degli attributi, perché risultano uguali ai nomi degli attributi corrispondenti nelle tabelle di definizione:

V1:
```
CREATE VIEW WORKS_ON1
AS     SELECT FNAME, LNAME, PNAME, HOURS
       FROM EMPLOYEE, PROJECT, WORKS_ON
       WHERE SSN=ESSN AND PNO=PNUMBER;
```

V2:
```
CREATE VIEW DEPT_INFO(DEPT_NAME, NO_OF_EMPS, TOTAL_SAL)
AS     SELECT DNAME, COUNT (*), SUM (SALARY)
       FROM DEPARTMENT, EMPLOYEE
       WHERE DNUMBER=DNO
       GROUP BY DNAME;
```

Una vista è sempre aggiornata; se si modificano le tuple nelle tabelle base su cui è definita una vista, automaticamente la vista riflette i cambiamenti. Se la vista non serve più, si può utilizzare il comando DROP VIEW per eliminarla:

V1A: DROP VIEW WORKS_ON1;

PARTE 4

10 Dipendenze funzionali e normalizzazione per basi di dati relazionali

Ogni schema di relazione è composto da un certo numero di attributi, e lo schema di dati relazionale è composto da in certo numero di schemi di relazione. Abbiamo bisogno di misurare formalmente il perché un raggruppamento di attributi in uno schema di relazione possa essere meglio di un altro. Ci sono due livelli ai quali possiamo misurare questa "bontà". Il primo è il livello logico (o concettuale) – come gli utenti interpretano gli schemi di relazione e il significato dei loro attributi. L'avere buoni schemi di relazione a questo livello consente agli utenti di capire chiaramente il significato dei dati nelle relazioni, e perciò di formulare correttamente le loro interrogazioni. Il secondo è il livello d'implementazione (o fisico) – come le tuple in una relazione di base sono memorizzate e aggiornate. Questo livello si applica solo a schemi di relazione di base – che saranno memorizzati fisicamente come file – mentre al livello logico noi siamo interessati a schemi sia di relazioni di base sia di viste.

10.1 Linee guida informali di progettazione di schemi di relazione

Vengono presentate qui quattro misure informali di qualità per la progettazione di uno schema di relazione:

1. Semantica degli attributi
2. Riduzione dei valori ridondanti nelle tuple
3. Riduzione del numero di valori nulli nelle tuple
4. Impossibilità di generare tuple spurie.

10.1.1 Semantica degli attributi di una relazione

Ogni volta che si raggruppano degli attributi per formare uno schema di relazione, si suppone che ad essi sia associato un certo significato. Ogni relazione può essere interpretata come un insieme di fatti e di asserzioni. Questo significato, o semantica, specifica come interpretare i valori degli attributi memorizzati in una tupla della relazione – in altre parole, come i valori degli attributi in una tupla sono in relazione tra loro. Se è stata eseguita correttamente la progettazione concettuale, seguita da una traduzione nelle relazioni, la maggior parte della semantica dovrebbe essere spiegata e il progetto risultante dovrebbe avere un significato chiaro.

Linee guida 1: si progetti ogni schema di relazione in modo tale che sia semplice spiegarne il significato. Non si uniscano attributi provenienti da più tipi di entità e tipi di associazione in un'unica relazione. Intuitivamente, se uno schema di relazione corrisponde a un solo tipo di entità o a un solo tipo di associazione, il suo significato tende ad essere chiaro. Altrimenti la relazione corrisponde ad un miscuglio di più entità e associazioni e perciò diventa semanticamente oscura.

10.1.2 Informazioni ridondanti nelle tuple e anomalie di aggiornamento

Uno scopo della progettazione di schemi è quello di ridurre al minimo lo spazio di memoria occupato dalle relazioni di base (file). Il raggruppamento di attributi in schemi di relazione ha un effetto significativo sullo spazio di memoria.

Un altro grave problema è il problema delle anomalie di aggiornamento, Esse possono essere classificate in anomalie di inserimento, anomalie di cancellazione o anomalie di modifica:

Anomalie di inserimento: possono essere di due tipi, illustrati dai seguenti esempi basati sulla relazione IMP_DIP

- Per inserire una nuova tupla impiegato in IMP_DIP occorre inserire i valori degli attributi del dipartimento in cui l'impiegato lavora, o valori nulli (se l'impiegato ancora non lavora per un dipartimento). Ad esempio, per inserire una nuova tupla per un impiegato che lavora nel dipartimento numero 5, bisogna inserire correttamente i valori degli attributi del dipartimento

5 in modo che siano consistenti con i valori del dipartimento 5 in altre tuple di IMP_DIP. Se le cose fossero staccate come nello schema precedente non ci si dovrebbe preoccupare di questo problema di consistenza, perché tutti gli altri valori del dipartimento 5 sono registrati una volta sola della BdD, come tupla singola della relazione DIPARTIMENTO.

- È difficile inserire nella relazione IMP_DIP un nuovo dipartimento che non ha ancora impiegati

Anomalie di cancellazione: questo problema è legato alla seconda situazione di anomalia di inserimento discussa sopra. Se si cancella da IMP_DIP una tupla impiegato che è quella che rappresenta l'ultimo impiegato che lavora per un dipartimento, l'informazione riguardante quel dipartimento non è più presente nella base di dati.

Anomalie di modifica: in IMP_DIP, se si cambia il valore di uno degli attributi di un particolare dipartimento occorre aggiornare di conseguenza tutte le tuple, altrimenti la BdD è inconsistente.

Linee Guida 2: si progettino gli schemi di relazione di base in modo che nelle relazioni non siano presenti anomalie di inserimento, cancellazione o modifica. Se sono presenti delle anomalie, le si rilevi chiaramente e ci si assicuri che i programmi che aggiornano le BdD operino correttamente.

10.1.3 Valori nulli nelle tuple

È possibile che nei progetti di alcuni schemi vengano raggruppati numerosi attributi a formare una grossa relazione. Se molti di questi attributi non riguardano tutte le tuple della relazione, quelle tuple finiscono con l'avere molti valori nulli. Ciò può dar luogo a uno spreco di spazio di memoria e può anche portare a problemi di comprensione del significato degli attributi e di specificazione delle operazioni di join a livello logico. Un altro problema con i valori nulli è come rispondere della loro presenza quando vengono usate funzioni aggregate come COUNT o SUM. Inoltre i valori nulli possono avere più interpretazioni, tra cui le seguenti:

- L'attributo non è pertinente per questa tupla
- Il valore dell'attributo per questa tupla è sconosciuto
- Il valore è noto ma assente, cioè non è ancora stato memorizzato.

L'uso della stessa rappresentazione per tutti i valori nulli non consente di cogliere i diversi significati che essi possono assumere.

Linee Guida 3: per quanto possibile, si eviti di porre in relazione di base attributi i cui valori possono essere frequentemente nulli. Se i valori nulli sono inevitabili, ci si assicuri che essi si presentino solo in casi eccezionali e che non riguardino una maggiore di tuple nella relazione.

10.1.4 Generazione di tuple spurie

Se con l'uso del join di due schemi di relazione si generano un numero maggiore di tuple di quelle desiderate si dice che ha generato tuple spurie, perché rappresentano un'informazione spuria o sbagliata che non è valida.

Linee guida 4: si progettino schemi di relazioni in modo tale che essi possano essere riuniti, tramite JOIN, con condizioni di uguaglianza su attributi che sono chiavi primarie o chiavi esterne in modo da garantire che non vengano generate tuple spurie. Non si abbiano relazioni che contengono attributi di accoppiamento diversi dalle combinazioni chiave esterna – chiave primaria. Se relazioni di questo tipo sono inevitabili, non si effettui su di esse un'operazione di Join sulla base di questi attributi, perché il join può produrre tuple spurie.

10.2 Dipendenze funzionali

10.2.1 Definizione di dipendenza funzionale

Una dipendenza funzionale è un vincolo tra due insiemi di attributi della BdD. Si supponga che il nostro schema di BdD relazione abbia n attributi $A_1,A_2,\ldots,A_n$: si pensi all'intera BdD come se fosse

descritta da un solo schema di relazione universale R = { A_1,A_2,…,A_n}. Ciò non significa che si memorizzerà effettivamente la BdD con una sola tabella universale; questo concetto verrà usato solo per sviluppare la teoria formale delle dipendenze tra dati.
Una dipendenza funzionale, indicata con X → Y, tra due insiemi di attributi X e Y che siano sottoinsiemi di R specifica un vincolo sulle tuple che possono formare uno stato di relazione r di R. Il vincolo è che coppia di tuple t1, t2 con t1[X] = t2[X] risulta anche t1[Y] = t2[Y]. Ciò significa che i valori della componente Y di una tupla in r dipendono da, o sono determinati da, i valori della componente X, o, in alternativa, che i valori della componente X di una tupla determinano univocamente (o funzionalmente) i valori della componente Y. Si dice che c'è una dipendenza funzionale da X a Y o che Y è funzionalmente dipendente da X.
Pertanto X determina funzionalmente Y in uno schema di relazione R se e solo se, ogni volta che due tuple di r(R) concordano sul loro valore X, essi devono necessariamente concordare anche sul loro valore Y. Si noti quanto segue:

- Se un vincolo su R stabilisce che no ci possa essere più di una tupla con un dato valore per X in una generica istanza di relazione r(R) – cioè X è una chiave candidata di R – ciò implica che X → Y per qualsiasi sottoinsieme Y di attributi di R (perché il vincolo di chiave implica che nessuna coppia di tuple in un qualsiasi stato valido r(R) avrà lo stesso valore di X).
- Se X → Y in R, ciò non ci dice se in R Y → X oppure no.

La dipendenza funzionale è una proprietà della semantica o del significato degli attributi. I progettisti della base di dati useranno la loro conoscenza della semantica degli attributi di R – vale a dire come sono collegati tra loro – per specificare le dipendenze funzionali che dovrebbero sussistere su tutti gli stati di relazione (estensioni) r di R. Ogni volta che la semantica di due insiemi di attributi in R indica che deve sussistere una dipendenza funzionale, la dipendenza viene specificata con un vincolo. Estensioni di relazione r(R) che soddisfano i vincoli di dipendenza funzionale sono dette estensioni valide di R, perché soddisfano i vincoli di dipendenza funzionale.
L'uso principale delle dipendenze funzionali è perciò quello di descrivere ulteriormente uno schema di relazione R, tramite una specificazione dei vincoli sui suoi attributi che devono valere sempre.
Certe dipendenze funzionali possono essere specificate senza riferirsi a una particolare relazione, ma come una proprietà degli attributi coinvolti. Ad esempio `{Stato, numero_patente}→SSN` deve valere per ogni adulto negli Stati uniti.
Si consideri lo schema IMP_PROG: dalla semantica degli attributi sappiamo che devono sussistere le seguenti dipendenze funzionali:

1. SSN → NOME_I
2. NUMERO_P → {NOME_P, SEDE_P}
3. {SSN, NUMERO_P} → ORE

10.2.2 Regole di inferenza per dipendenze funzionali

Si indichi con F l'insieme di dipendenze funzionali specificate sullo schema di relazione R. Tipicamente il progettista dello schema specifica le dipendenze funzionali semanticamente evidenti; di solito, però, sussistono molte altre dipendenze funzionali in tutte le istanze di relazione valide che soddisfano le dipendenze in F, Queste altre dipendenze possono essere inferite o dedotte dalle dipendenze funzionali presenti in F. l'insieme di tutte queste dipendenze è detto chiusura di F ed è indicato con F^+. Ad esempio, si supponga si specificare il seguente insieme F di dipendenze funzionali sullo schema di relazione seguente:

FIGURA 10.3A

F={ SSN → {NOME_I, DATA_N, INDIRIZZO, NUMERO_D},
NUMERO_D → {NOME_D, SSN_DIR_DIP}}

È possibile inferire da F le seguenti dipendenze funzionali aggiuntive:

SSN → {NOME_D, SSN_DIR_DIP}
SSN → SSN
NUMERO_D → NOME_D

Una dipendenza funzionale (DF) X→Y è inferita da un insieme di dipendenze F specificate su R se X→Y sussiste in ogni stato di relazione r che sia un'estensione valida di R; cioè ogni volta che r soddisfa tutte le dipendenze in F, sussiste anche X→Y. La chiusura F^+ di F è l'insieme di tutte le dipendenze funzionali che possono essere dedotte da F. Per determinare un modo sistematico per inferire dipendenze, occorre trovare un insieme di regole di inferenza che possono essere usate per inferire nuove dipendenze a partire da un insieme dato di dipendenze. La notazione F|=X→Y indica che la dipendenza funzionale X→Y è inferita dall'insieme di dipendenze funzionali F. La DF {X,Y}→ {V,Z} viene abbreviata con XY→VZ
Le seguenti sei regole sono note come regole di inferenza per dipendenze funzionali
RI1 (regola riflessiva): se X⊇Y, allora X→Y
RI2 (regola di arricchimento): se {X→Y} |= XZ→YZ
RI3 (regola transitiva): se {X→Y, Y→ Z} |= X→Z
} Regole di inferenza di Armstrong
RI4 (regola di decomposizione, o di proiezione): {X→YZ} |= X→Y
RI5 (regola di unione, o additiva): {X→Y, X→Z} |= X→YZ
RI6 (regola pseudo-transitiva): {X→Y, WY→Z} |= WX→Z

Per ogni insieme X di attributi di questo tipo, si calcola l'insieme X^+ di attributi che sono determinati funzionalmente da X sulla base di F; X^+ è detto chiusura di X rispetto a F. Per calcolare X^+ si può usare l'algoritmo seguente:

$X^+ := X$;
repeat
    $oldX^+ := X^+$
    for ogni dipendenza funzionale Y→Z in F do
        if $X^+ \supseteq Y$ then $X^+ := X^+ \cup Z$;
until ($X^+ = oldX^+$);

10.2.3 Equivalenza di insiemi di dipendenze funzionali

Un insieme di dipendenze funzionali E è coperta da un insieme di dipendenze funzionali F – alternativamente si dice che F copre E – se ogni DF in E è presente anche F^+, cioè se ogni dipendenza in E può essere inferita in F. Due insiemi E ed F di dipendenze funzionali sono equivalenti se $E^+=F^+$. Perciò l'equivalenza implica che ogni DF in E possa essere inferita in F, e ogni DF in F possa essere inferita da E, ossia E è equivalente a F se sussistono le condizioni E copre F e F copre E.
Si può determinare se F copre E calcolando X^+ rispetto a F per ogni DF X→Y in E, e quindi verificando se questo X^+ comprende gli attributi presenti in Y. Se è così per ogni DF in E, allora F copre E. Si determina se E e F sono equivalenti verificando se E copre F e F copre E.

10.2.4 Insiemi minimali di dipendenze funzionali

Un insieme F di dipendenze funzionali è minimale se soddisfa le seguenti condizioni:
1. ogni dipendenza presente in F ha come parte destra un solo attributo
2. non è mai possibile sostituire una dipendenza X→A di F con una dipendenza Y→A, dove Y è un sottoinsieme proprio di X, e avere ancora un insieme di dipendenze equivalente a F
3. Non è possibile rimuovere una dipendenza da F e avere ancora un insieme di dipendenze equivalente

Si può pensare a un insieme minimale di dipendenze come a un insieme di dipendenze in forma standard o canonica senza ridondanze. Una copertura minimale di un insieme F di dipendenze funzionali è un insieme minimale di dipendenze F_{min} equivalente ad F. Purtroppo per uno stesso insieme di dipendenze funzionali ci possono essere molte coperture minimali. Usando il seguente algoritmo si può sempre trovare almeno una copertura minimale G per ogni insieme F di dipendenze:

1. Poni G :=F
2. Sostituisci ogni dipendenza funzionale $X \rightarrow \{A_1, A_2, .., A_n\}$ in G con n dipendenze funzionali $X \rightarrow A_1$, $X \rightarrow A_2$,…,$X \rightarrow A_n$
3. Per ogni dipendenza funzionale $X \rightarrow A$ in G
   Per ogni attributo B che sia un elemento di X se
   $((G - \{X \rightarrow A\}) \cup \{(X - \{B\}) \rightarrow A\})$ è equivalente a G,
   allora sostituisci $X \rightarrow A$ con $(X - \{B\}) \rightarrow A$ in G
4. Per ogni dipendenza funzionale rimanente $X \rightarrow A$ in G se $(G - \{X \rightarrow A\})$ è equivalente a G, allora rimuovi $X \rightarrow A$ da G

10.3 Forme normali basate su chiavi primarie

10.3.1 Introduzione alla normalizzazione

Il processo di normalizzazione sottopone una schema di relazione a una serie di test per certificare se soddisfa a una certa forma normale. Il processo, che procede in modo top-down valutando ogni relazione coni criteri per le forme normali e decomponendo le relazioni quando necessario, può pertanto essere considerato come una progettazione relazionale per analisi. Sono state create 5 forme normali e la forma normale di Boyce e Codd.
La normalizzazione dei dati può perciò essere considerata come un processo di analisi degli schemi di relazione forniti, basato sulle loro dipendenze funzionali e chiavi primarie per raggiungere le proprietà desiderate di (1) minimizzazione della ridondanza, (2) minimizzazione delle anomalie di inserimento, cancellazione e modifica. Schemi di relazione inadeguati, che non soddisfano certe condizioni – i test di forma normale – vengono decomposti in schemi di relazione più piccoli che superano i test e pertanto possiedono le proprietà desiderate. Quindi la procedura di normalizzazione fornisce ai progettisti di BdD:

- Una struttura formale di analisi degli schemi di relazione basata sulle loro chiavi e sulle dipendenze funzionali tra i loro attributi
- Una serie di test in forma normale che possono essere effettuati sui singoli schemi di relazione, in modo che la BdD relazione possa essere normalizzata fino al livello desiderato, qualunque esso sia.

La forma normale di una relazione fa riferimento alla più alta condizione di forma normale soddisfatta, e perciò indica il livello al quale è stata normalizzata. Quando vengono considerate separatamente da altri fattori, le forme normali non garantiscono una buona progettazione di basi di dati. Di solito non è sufficiente verificare separatamente che ogni schema di relazione nella BdD sia, ad esempio in BCNF o in 3NF. Il processo di normalizzazione tramite decomposizione deve piuttosto confermare anche l'esistenza di proprietà aggiuntive con gli schemi di relazione, considerati nel loro insieme, devono possedere, come:

- La proprietà di join senza perdita (lossless join) o di join non additivo (nonadditive join), che garantisce che negli schemi di relazione creati dopo una decomposizione non si presenti il problema di generazione di tuple spurie
- La proprietà di conservazione delle dipendenze (dependency preservation), che assicura che ogni dipendenza funzionale sia rappresentata in una qualche relazione risultante dopo la decomposizione.

La proprietà di join non additivo è estremamente critica e deve essere raggiunta a qualsiasi costo, mentre la proprietà di conservazione delle dipendenze, sebbene desiderabile, viene qualche volta sacrificata.
Un altro punto degno di nota è che i progettisti di BdD non sono obbligati a normalizzare fino alla più alta forma normale possibile. Le relazioni possono essere lasciate ad un livello più basso di normalizzazione per ragioni di prestazioni. Il processo di memorizzazione del join di relazioni in forma normale più alta come una relazione di base – in forma normale più bassa – è noto come denormalizzazione.
Una superchiave di uno schema di relazione $R=\{A_1, A_2, \ldots, A_n\}$ è un insieme di attributi $S \subseteq R$ con la proprietà che nessuna coppia di tuple t_1 e t_2 in un generico stato valido di relazione r di R avrà $t_1[S]=t_2[S]$. Una chiave K è una superchiave con la proprietà aggiuntiva che la rimozione di un qualsiasi attributo da K fa sì che K cessi di essere superchiave. La differenza tra una chiave e una superchiave sta nel fatto che una chiave deve essere minimale.
Se uno schema di relazione ha più di una chiave, ognuna di esse è detta chiave candidata. Una delle chiavi candidate è arbitrariamente nominata chiave primaria, e le altre sono dette chiavi secondarie.
Un attributo di uno schema di relazione R è detto attributo primo di R se esso è membro di una qualche chiave candidata di R. Un attributo è detto non primo se non è un attributo primo – cioè se non è membro di nessuna chiave candidata.

10.3.2 Prima forma normale

La prima forma normale 1NF è ora considerata parte integrante della definizione formale di relazione nel modello relazionale di base. Essa richiede che il dominio di un attributo comprenda solo valori atomici (semplici, indivisibili) e che il valore di qualsiasi attributo in una tupla sia un valore singolo del dominio di quell'attributo. Perciò la 1NF non consente di avere un insieme di valori, una tupla di valori o una combinazione di entrambi come valore attributo per uan tupla singola. In altre parole, la 1NF non permette "relazioni entro relazioni" o "relazioni come attributi di tuple". I soli valori di attributi consentiti dalla 1NF sono i singoli valori atomici (o indivisibili).
Esempio:
IMMAGINI 10.8
Ci sono tre tecniche principali per raggiungere la prima forma normale per una relazione di questo tipo:

1. si rimuova l'attributo Sedi_D che viola la 1NF e lo si ponga in una relazione separata Sedi_dip insieme con la chiave primaria NUMERO_D e DIPARTIMENTO. La chiave primaria di questa relazione è la combinazione{NUMERO_D, SEDE_D}. In SEDI_DIP esiste una tupla distinta per ogni sede di un dipartimento. Ciò decompone la relazione che non è in 1NF in due relazioni in 1NF
2. Si espanda la chiave in modo tale che si sia una tupla separata nella relazione DIPARTIMENTO originaria per ogni sede di un DIPARTIMENTO. In questo caso la chiave primaria diventa la combinazione {NUMERO_D, SEDE_D}. Questa soluzione ja lo svantaggio di introdurre ridondanza nella relazione.
3. Se è noto il numero massimo di valori dell'attributo – ad esempio ogni dipartimento non può avere più di tre sedi – si sostituisce SEDI_DIP con SEDE_DIP1, SEDE_DIP2 e SEDE_DIP3. Questa soluzione ha lo svantaggio di introdurre valori nulli se la maggior parte dei dipartimenti ha meno di tre sedi.

La prima forma normale non permette neppure attributi multivalore che siano essi stessi composti. Queste relazioni sono dette relazioni nidificate perché ogni tupla può avere al suo interno una relazione.
Nella figura seguente
FIGURA 10.9
Viene mostrato come potrebbe apparire la relazione IMP_PROG se fosse consessa la nidificazione. Ogni tupla rappresenta un'entità impiegato, e una relazione PROGG(NUMERO_P, ORE)

all'interno di ogni tupla rappresenta i progetti su cui lavora l'impiegato e le ore settimanali lavorate dall'impiegato su ogni progetto. Lo schema di questa relazione IMP_PROG può essere rappresentato come segue:
IMP_PROG(SSN, NOME_I, {PROGG(NUMERO_P, ORE)})
Le parentesi graffe {} indicano che l'attributo PROGG è nultivalore, mentre gli attributi componenti che formano PROGG sono posti tra parentesi ().
SSN è la chiave primaria della relazione IMP_PROG mentre NUMERO_P è la chiave parziale della relazione nidificata; cioè la relazione nidificata deve avere valori univoci per NUMERO_P, all'interno di ogni tupla. Per normalizzare questa relazione in 1NF si spostano gli attributi della relazione nidificata in una nuova relazione e si propaga la chiave primaria al suo interno; la chiave primaria della nuova relazione combinerò la chiave parziale con la chiave primaria della relazione originaria. La decomposizione e la propagazione della chiave primaria producono gli schemi IMP_PROG1 e IMP_PROG2 seguenti:

IMP_PROG1

SSN	NOME_I

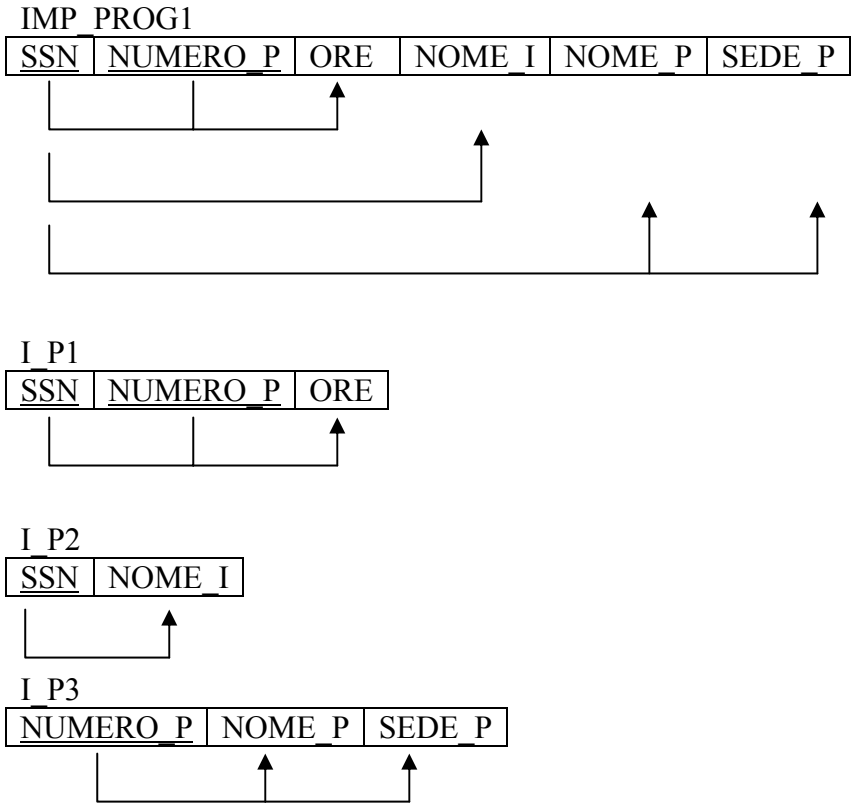
IMP_PROG2

SSN	NOME_I	ORE

Questa procedura può essere applicata ricorsivamente a una relazione con nidificazione a più livelli per eliminare la nidificazione (unnest) della relazione, portandola ad un insieme di relazioni in 1NF. Ciò è utile per convertire uno schema di relazione non normalizzato con molti livelli di nidificazione in relazioni in 1NF. L'operatore unnest fa parte del modello relazionale nidificato.

10.3.3 Seconda forma normale

La seconda forma normale 2NF si basa sul concetto di dipendenza funzionale completa. Una dipendenza funzionale X→Y è una dipendenza funzionale completa se la rimozione di qualsiasi attributo A da X comporta che la dipendenza non sussista più; cioè, per ogni attributo $A \in X$, (X–{A}) non determina funzionalmente Y. Una dipendenza funzionale X→Y è una dipendenza parziale se si possono rimuovere da X certi attributi $A \in X$ e la dipendenza continua a sussistere; cioè, per qualche $A \in X$, (X – {A})→Y.

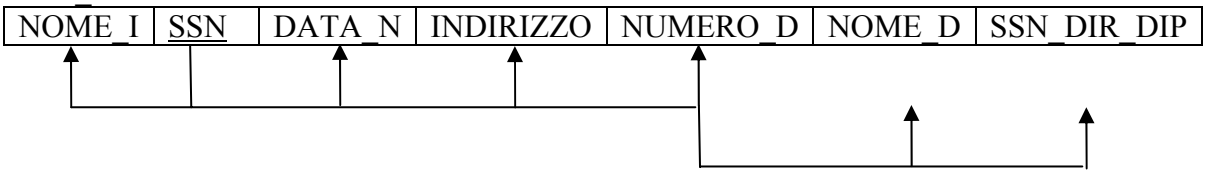


Il test per la seconda forma normale comporta l'esame di dipendenze funzionali i cui attributi di parte sinistra fanno parte della chiave primaria. Se la chiave primaria contiene un solo attributo è inutile effettuare il test. Uno schema di relazione R è in 2NF se ogni attributo non primo A di R dipende funzionalmente in modo completo dalla chiave primaria di R. Se uno schema di relazione non è in 2NF, esso può essere normalizzato in 2NF componendolo in un certo numero di relazioni in 2NF, nelle quali gli attributi non primi sono associati solo alla parte della chiave primaria da cui sono funzionalmente dipendenti in modo completo.

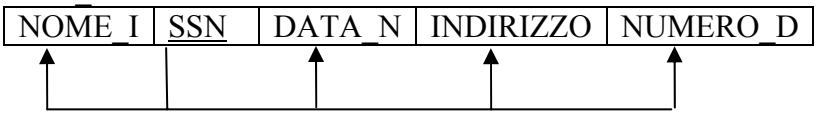
10.3.4 Terza forma normale

La terza forma normale 3NF è basata sul concetto di dipendenza transitiva. Una dipendenza funzionale X→Y in uno schema di relazione R è una dipendenza transitiva se esiste un insieme di attributi Z, che non è né chiave candidata né un sottoinsieme di una chiave di R, per cui valgono contemporaneamente X→Z e Z→Y. Ad esempio il seguente schema è in 2NF ma non in 3NF, a causa della dipendenza transitiva di SSN_DIR_DIP (e anche NOME_D) da SSN attraverso NUMERO_D:

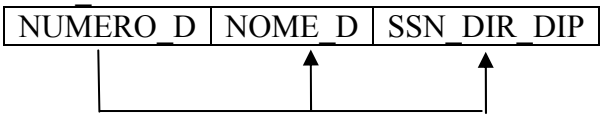
IMP_DIP



IMP_D1



IMP_D2



Si pu; normalizzare IMP?DIP decomponendolo nei due schemi di relazione in 3NF IMP_D1 e IMP_D2.

Forma	Test	Rimedio (normalizzazione)
1NF	La relazione non deve avere attributi non-atomici o relazioni nidificate	Formare nuove relazioni per ogni attributo non-atomico o relazioni nidificate
2NF	Per relazioni in cui la chiave primaria contiene più attributi, nessun attributo non-chiave deve essere funzionalmente dipendente da una parte della chiave primaria	Decomporre e preparare una nuova relazione per ogni chiave parziale con i suoi attributi dipendenti. Assicurarsi di mantenere una relazione con la chiave primaria originale e tutti gli attributi funzionalmente dipendenti in modo completo da essa
3NF	La relazione non deve avere un attributo non-chiave determinato funzionalmente da un altro attributo non-chiave (o da un insieme di attributi non chiave). Cioè non deve esserci nessuna dipendenza transitiva di un attributo non-chiave della chiave primaria	Decomporre e preparare una relazione che comprenda gli attributi non-chiave che determinano funzionalmente altri attributi non-chiave.

10.5 Forma normale di Boyce e Codd

La forma normale di Boyce e Codd (BCNF) è stata proposta come una forma più semplice di 3NF, ma si è rivelata più restrittiva della 3NF; infatti, ogni relazione in BCNF è anche in 3NF ma una relazione in 3NF non è detto che sia in forma normale. La definizione formale di BCNF differisce solo leggermente dalla definizione di 3NF. Uno schema di relazione R è in BCNF se, ogni volta che sussiste una dipendenza funzionale non banale X→A, X è una superchiave di R.
In pratica la maggior parte degli schemi di relazione in 3NF è anche in BCNF, Solo se in uno schema di relazione R sussiste X→A, con X che non è superchiave e A attributo primo, R sarà in 3NF ma non in BCNF.

PARTE 5

5.1 Introduzione

La collezione di dati che costituisce una BdD computerizzata deve essere fisicamente memorizzata su un qualche supporto di memoria del calcolatore. Il software del DBMS poi potrà recuperare, aggiornare ed elaborare questi dati quando necessario. I supporti di memoria del calcolatore formano una gerarchia di memoria che comprende due categorie fondamentali:

- Memoria principale: questa categoria comprende supporti di memoria su cui può operare direttamente la CPU, come i registri della cpu e la memoria cache.
- Memoria secondaria: questa categoria comprende dischi magnetici, dischi ottici e nastri. Questi dispositivi hanno solitamente una maggiore capacità, costano meno ma forniscono un accesso ai dati più lento.

5.1.2 Memorizzazione di basi di dati.

Le BdD tipicamente memorizzano grandi quantità di dati, i quali devono persistere per lunghi periodi di tempo. Durante questi periodi si accede ai dati e li si elabora ripetutamente. Ciò è in contrasto con la nozione di strutture transitorie che persistono solo per un tempo limitato durante l'esecuzione del programma. La maggior parte delle BdD è memorizzata permanentemente (o persistentemente) su memoria secondaria a disco magnetico, per le seguenti ragioni:

- Di solito le BdD sono troppo ampie per poter essere interamente contenute in memoria centrale.
- Le circostanze che causano una perdita permanentemente di dati memorizzati si verificano meno frequentemente per la memorizzazione secondaria su disco che per la memorizzazione primaria.
- Il costo di memorizzazione per unità di dato è di un ordine di grandezza inferiore per il disco che per la memoria principale

Le applicazioni tipiche di BdD hanno bisogno per l'elaborazione solo di una piccola porzione alla volta della base di dati. Tutte le volte che c'è bisogno di una certa porzione dei dati, essa deve essere localizzata su disco, copiata in memoria centrale per l'elaborazione, e quindi riscritta su disco se i dati sono cambiati. I dati memorizzati su disco sono organizzati come file di record. Ogni record è costituito da una collezione di valori di dati che possono essere interpretati come fatti relativi alle entità, ai loro attributi e alle loro associazioni. I record dovrebbero essere memorizzati su disco in modo da rendere possibile individuare efficientemente la collocazione ogni volta che se ne ha bisogno.
Ci sono molte organizzazioni primarie dei file che determinano come sono collocati fisicamente sul disco i record di un file, e perciò come si può accedere ad essi. Un file heap (heap: ammasso, cumulo) (o file non ordinato) colloca i record su disco senza un ordine particolare, semplicemente aggiungendo nuovi record alla fine del file, mentre un file ordinato (o sequenziale) tiene i record in ordine sulla base del valore di un campo particolare (detto chiave di ordinamento). Un file hash (Hash: tritare, sminuzzare) usa una funzione hash applicata a un campo particolare (detto chiave hash) per determinare la collocazione di un record su disco. Altre organizzazioni primarie dei file, come i B-alberi, usano appunto strutture ad albero. Un'organizzazione secondaria, o struttura di accesso ausiliaria, consente un accesso efficiente ai record di un file basata su campi alternativi rispetto a quelli che sono stati usati per l'organizzazione primaria dei file.

5.4 Bufferizzazione dei blocchi

Quando devono essere trasferiti da disco a memoria centrale numerosi blocchi, e sono noti tutti i loto indirizzi, per accelerare il trasferimento possono essere riservati in memoria centrale molti buffer. Mentre viene letto o scritto un buffer, la CPU può elaborare i dati nell'altro buffer. Ciò è possibile perché esiste un processore indipendente per l'I/O di disco che può lavorare in parallelo con la CPU.

5.5 Collocazione su disco dei record di un file

5.5.1 Record e tipi di record

I dati sono normalmente memorizzati sotto forma di record. Ogni record consiste di una collocazione di valori o voci di dati collegati, dove ogni valore è formato da uno o più byte e corrisponde ad un particolare campo del record. I record di solito descrivono le entità e i loro attributi. Una collezione di nomi di campi e dei tipi di dati corrispondenti costituisce una definizione di tipo di record o formato di record. Un tipo di dati, associato a ciascun campo, specifica il tipo di valori che possono essere assunti da quel campo.
In applicazioni di basi di dati recenti può sorgere la necessità di memorizzare voci di dati che consistono i grandi oggetti non strutturati, rappresentati immagini, flussi digitalizzati audio o video, o testo libero. Queste sono indicate come BLOB (Binary Large OBject); una voce di dati BLOB è tipicamente memorizzata separatamente dal suo record in un pool di blocchi di disco, e ci si limita ad inserire un puntatore al BLOB.

5.5.2 File, record a lunghezza fissa e record a lunghezza variabile

Un file è una sequenza di record. In molti casi tutti i record presenti in un file fanno parte dello stesso tipo di record. Se ogni record nel file ha esattamene la stessa dimensione (in byte), si dice che il file è costituito da record a lunghezza fissa. Se record diversi nel file hanno dimensioni diverse, si dice che il file è costituito da record a lunghezza variabile. Un file può avere record a lunghezza variabile per molte ragioni:

- I record del file sono dello stesso tipo di record, ma uno o più campi sono di dimensione variabile (campi di lunghezza variabile), ad esempio un campo Nome di Impiegato.
- I record del file sono dello stesso tipo di record, ma uno o più campi può avere valori multipli per singoli record; un tale campo è detto campo che si ripete (repeating field) e un gruppo di valori per quel campo è detto gruppo che si ripete.
- I record del file sono dello stesso tipo di record, ma uno o più campi sono opzionali.
- Il file contiene record di diversi tipi di record e perciò di dimensione variabile (file misto). Ciò potrebbe verificarsi se records collegati di tipi diversi fossero raggruppati su blocchi del disco.

Se ogni record ha gli stessi campi, e le lunghezze dei campi sono fisse, il sistema può individuare la posizione del byte di inizio di ciascun campo facilmente. È possibile rappresentare un file che da un punto di vista logico dovrebbe avere record a lunghezza variabile come un file di record a lunghezza fissa (campi opzionali → comunque presenti con valore null se vuoti, per un campo che si ripete avere un massimo e memorizzarli tutti, al limite vuoti…).
Per campi di lunghezza variabile ogni record presenta un valore in corrispondenza a ogni campo, ma non è nota la lunghezza esatta dei valori di alcuni campi. Per determinare i bye che all'interno di un particolare record rappresentano ogni campo, si possono usare speciali caratteri separatori (come ?, % o $) – che non si presenta in nessun valore del campo – per terminare campi di lunghezza variabile, oppure memorizzare la lunghezza in byte del campo in quel record, prima del valore del campo.
Un file di record con campi opzionali può essere formattato in diversi modi. Se il numero totale di campi per il tipo di record è grande, ma il numero di campi che si presentano effettivamente in un

record è piccolo, è possibile inserire in ciascun record una sequenza di coppie <nome-campo, valore-campo> piuttosto che solo i valori del campo.

5.5.3 Ripartizione dei record in blocchi e confronto tra record con spanning e record senza spanning

I record di un file devono essere ripartiti su blocchi di disco perché un blocco è l'unità si trasferimento dati tra disco e memoria. Quando la dimensione del blocco è maggiore di quella del record, ogni blocco conterrà numerosi record, anche se alcuni file possono avere record inusualmente grandi che non possono trovar posto in un solo blocco. Si supponga che la dimensione del blocco dia di B byte. Per un file di record a lunghezza fissa della dimensione di R byte, con $B \geq R$, è possibile inserire $bfr = \lfloor B/R \rfloor$ record per blocco. Il valore bfr è detto fattore di blocco (blocking factor) per il file. In generale R può non dividere B esattamente, cosichè in ogni blocco si ha un certo spazio inutilizzato, pari a:

$$B - (bfr * R) \text{ byte}$$

Per utilizzare questo spazio si può memorizzare parte di un record in un blocco e il resto in un altro. Un puntatore alla fine del primo blocco punta al blocco che contiene il resto del record nel caso in cui non sia il blocco consecutivo su disco. Questa organizzazione è detta spanned (estesa), perché o record possono estendersi su più di un blocco. Ogni volta che un record è più grande di un blocco si deve usare un'organizzazione spanned. Se ai record non è concesso di attraversare i confini di un blocco, l'organizzazione è detta unspanned. Essa è usata con record a lunghezza fissa che hanno $B > R$, perché consente a ciascun record di cominciare in una locazione nota del blocco, semplificando l'elaborazione dei record. Per record a lunghezza variabile può essere usata sia un'organizzazione spanned sia un'organizzazione unspanned. Se il record tipico è grande, è vantaggioso usare l'organizzazione con spannino.
Per record a lunghezza variabile che usano l'organizzazione spanned, ogni blocco può memorizzare un diverso numero di record. In questo caso il fattore di blocco bfr rappresenta il numero medio di record per blocco per il file. Si può usare bfr per calcolare il numero medio di blocchi b necessario per un file di r record:

$$b = \lceil (r/bfr) \rceil \text{ blocchi}$$

5.5.4 Allocazione dei blocchi di un file su disco

Ci sono molte tecniche standard per allocare i blocchi di un file su disco. Nell'allocazione contigua i blocchi dei file sono allocati su blocchi del disco consecutivi: ciò rende molto veloce la lettura dell'intero file con l'uso della doppia bufferizzazione, ma nel contempo rende difficile l'aumento delle dimensioni del file. Nell'allocazione concatenata ogni blocco contiene un puntatore al successivo blocco del file: ciò facilita l'aumento delle dimensioni del file ma rallenta la lettura dell'intero file. Una combinazione delle due alloca cluster di blocchi di disco consecutivi, con i cluster collegati fra loro. I cluster sono talora detti segmenti del file o estensioni del file. Un'altra possibilità è quella di usare un'allocazione indicizzata, dove uno o più blocchi di indici contengono puntatori agli effettivi blocchi dei file.

5.5.5 Header dei file

Un header di file (header: intestazione) o descrittore di file contiene informazioni su un file, e comprende le informazioni per determinare gli indirizzi di disco dei blocchi del file, nonché le descrizioni dei formati dei record, che possono comprendere:

- Fixed-lenght unspanned record: la lunghezza dei campi, l'ordine dei campi del record
- Variable length: codici tipo dei campi, caratteri di separazione, codici di tipo dei records.

Per la ricerca di un record su disco, uno o più blocchi sono copiati nei buffer di memoria centrale. Quindi i programmi ricercano il record o i record desiderati entro i buffer, usando le informazioni presenti nell'header del file. Se l'indirizzo del blocco che contiene il record desiderato non è noto, i programmi di ricerca devono fare una ricerca lineare attraverso i blocchi del file. Ogni blocco del file è copiato in un buffer ed esaminato fino a che il record è localizzato o tutti i blocchi del file sono stati esaminati senza successo. Ciò può essere molto dispendioso in termini di tempo per un file molto grande. Lo scopo di una buona organizzazione di file è quello di localizzare il blocco che contiene il record desiderato con il minimo numero di trasferimenti di blocco.

5.6 operazioni sui file.

Le operazioni sui file si possono dividere in due categorie: operazioni di retrieval e operazioni di update.
Retrieval: non cambiano alcun dato nel file, ma localizzano solo alcuni records in modo tale da poterne esaminare e processare i field values
Update: modificano il file inserendo o cancellando records o modificandone alcuni field values.
In entrambi i casi è necessario selezionare uno o più records per retrieval, deletion o modifica sulla base di una select condition. Una semplice condizione di selezione può essere rappresentata da operatori di =, <, >, … su alcuni campi.
Le operazioni di ricerca su file sono generalmente basate su condizioni di selezione semplici. Una condizione complessa deve essere decomposta dal DBMS per estrarre una condizione semplice che possa essere usata per localizzare i record su disco. Ogni record selezionato viene poi esaminato per decidere se soddisfa l'intera condizione di selezione.
Quando molti record di un file soddisfano una condizione di ricerca, viene generalmente localizzato il primo record – relativamente alla sequenza fisica dei record – nominato record corrente Le operazioni di ricerca seguenti cominciano da questo record e localizzano il successivo record del file che soddisfa la condizione.
Le operazioni effettive per localizzare e accedere ai record di un file generalmente sono le seguenti:

- *Open*: prepara il file per la lettura/scrittura. Alloca buffer appropriati, recupera l'header del file, imposta il file pointer.
- *Reset*: imposta il file pointer all'inizio del file.
- *Find (Locate)*: cerca il primo record che soddisfa una condizione di ricerca. Trasferisce il blocco in memoria primaria e lo setta come record corrente.
- *Read (get)*: copia il record corrente dal buffer in una variabile di programma del programma utente.
- *FindNext*: ricerca il successivo record nel file che soddisfa i criteri di ricerca, lo trasferisce in memoria principale e lo setta come record corrente.
- *Delete*: cancella il record corrente.
- *Modify*: modifica i valori di alcuni campi del record corrente.
- *Insert*: inserisce un nuovo record
- *Close*: completa l'accesso al file svuotando i buffer.

Le precedenti operazioni (tranne open e close) sono dette operazioni un-record-alla-volta, ma ci sono anche quelle definite un-insieme-alla-volta:

- *FindAll*: localizza tutti i record che soddisfano una condizione di ricerca.
- *FindOrdered:* recupera tutti i record ma anche in un ordine specificato
- *Reorganize:* ad esempio riordino.

Si definisce File organization l'organizzazione dei dat di un file in records, blocchi e strutture di accesso (ciò include il modo in cui records e blocchi sono memorizzati e interfacciati)
Si definisce Access methods un gruppo di operazioni che possono essere applicate sui file.

5.7 File di record non ordinati (file heap)

I record sono collocati nel file nell'ordine in cui sono inseriti, cosicché i nuovi records sono inseriti alla fine del file. Un'organizzazione di questo tipo è detta file heap o file pila. L'inserimenti di un nuovo record è molto efficiente: l'ultimo blocco di disco del file viene copiato in un buffer, viene aggiunto il nuovo record e poi il blocco viene riscritto sul disco. Però la ricerca di un record comporta una ricerca lineare sul file. Molto dispendioso. Se un solo record soddisfa la condizione di ricerca, allora, in media, un programma leggerà (portandoli in memoria) e ispezionerà metà dei blocchi di un file prima di trovare il record. Per un file costituito da b blocchi, ciò può richiedere in media (b/2) blocchi. Se nessun record soddisfa i criteri di ricerca bisognerà leggere tutti i b blocchi.
Per cancellare un record, un programma deve dapprima trovare il suo blocco, copiare il blocco in un buffer, quindi cancellare il record dal buffer e infine riscrivere il blocco su disco, ma sprecando quindi spazio. Altrimenti si può usare un bit come indicatore di cancellazione: quando si vuole cancellare quel record, si pone il bit a 1; la ricerca non tratterà i record con quel bit settato. In ogni caso entrambe le tecniche prevedono una periodica riorganizzazione del file per recuperare spazio.
Per un file non ordinato è possibile usare sia un'organizzazione spanned che unspanned; inoltre esso può essere usato con record a lunghezza fissa o con record a lunghezza variabile.
Per leggere tutti i record ordinati secondo i valori di un certo campo si può creare una copia ordinata del file. L'ordinamento è un'operazione dispendiosa per un grande file su disco; vengono per questo usate tecniche speciali per l'ordinamento esterno.
Per un file di record a lunghezza fissa non ordinati, che usa blocchi senza spanning e allocazione contigua, viene naturale accedere a ogni record a partire dalla sua posizione del file. Se i record del file sono numerati 0, 1, 2, …, r-1 e i record di ogni blocco sono numerati 0, 1, …, bfr-1, allora l'i-esimo record del file è posto nel blocco $\lfloor (i/bfr) \rfloor$ ed è l'(i mod bfr)-esimo record di quel blocco. Un file di questo tipo viene spesso chiamato file relativo o diretto perché si può agevolmente accedere direttamente ai record a partire dalle loro posizioni relative.

5.8 File di record ordinati (file sorted)

È possibile ordinare fisicamente i record di un file su disco basandosi sui valori di uno dei loro campi, detto campo di ordinamento. Ciò porta a un file ordinato, o file sequenziale. Se il campo di ordinamento è anche un campo chiave del file – un campo per cui è garantita l'esistenza di un valore univoco in ogni record – allora esso viene detto chiave di ordinamento del file.
I file di record ordinati presentano alcuni vantaggi sui file di record non ordinati. In primo luogo, la lettura dei record secondo l'ordine dei valori della chiave di ordinamento diventa estremamente efficiente, perché non è richiesta alcuna azione di riordinamento. In secondo luogo, trovare il record successivo a quello corrente, secondo l'ordine della chiave di ordinamento, di solito non richiede accessi aggiuntivi ai blocchi, perché il record successivo è nello stesso blocco di quello precedente. Infine, l'uso di una condizione di ricerca basata sul valore di un campo chiave di ordinamento ha come risultato un accesso più veloce di quando viene usata la tecnica di ricerca binaria, che sostituisce un miglioramento rispetto alle ricerche lineari.
La ricerca binaria può essere effettuata sui blocchi più che sui records: supponiamo di avere un file con b blocchi numerati 1,2,…, b i cui records siano ordinati secondo il valore crescente della loro chiave di ordinamento. Supponiamo di voler cercare un record il cui valore del campo chive sia K. Una ricerca binaria accede a $\log_2(b)$ blocchi sia che il record sia trovato o no, mentre una tecnica di ricerca lineare (in media) accede a b/2 blocchi se il record è trovato, a b blocchi se non è trovato.

Algoritmo di ricerca binario

```
l ← 1; u ← b; //b è il numero di blocchi del file
While (u ≥ l) do
      Begin i ← (l+u)/2;
      trasferisci il blocco i del file nel buffer;
```

```
      if K < (valore del campo chiave di ordinamento del primo record nel blocco
i)
      then u ← i-1
      else if K > (valore del campo chiave di ordinamento dell'ultimo record nel
blocco i)
      then l ← i+1
      else if il record con valore del campo chiave di ordinamento = K è nel
buffer
            then goto trovato
            else goto nontrovato;
      end;
goto nontrovato;
```

Un criterio di ricerca che coinvolge le condizioni <, >, ≥, ≤ su un campo ordinato è efficiente poiché tutti i record che soddisfano la condizione sono contigui nel file.
Le operazioni di inserzione e cancellazione di records sono operazioni costose per un file ordinato poiché il record deve rimanere fisicamente ordinato.
Inserzione: (a) cercare la sua posizione corretta nel file; (b) creare lo spazio nel file per inserire il nuovo file (mediamente bisogna spostare ½ file, cioè riscrivere metà dei blocchi di un file.
Soluzioni:

a) Tenere in ogni blocco dello spazio libero per eventuali inserimenti di record
b) Creare un file temporaneo non ordinato (overflow o transactional file) Nuovi records sono inseriti alla fine dell'overflow file. Periodicamente tale file è unito con il file principale durante l'operazione di riorganizzazione.

Modifica: modificare il valore di un record dipende da due fattori: (a) la condizione di ricerca e (b) il campo da modificare.
I file ordinati sono raramente usati nelle applicazioni di BdD se non è usato un cammino di accesso aggiuntivo, detto indice primario; ciò ha come risultato un file indicizzato-sequenziale. Questo migliora ulteriormente il tempo di accesso casuale sul campo chiave di ordinamento.

5.9 Tecniche hash

Un altro tipo di organizzazione primaria di file è basato sull'hash, che fornisce un accesso molto rapido ai record sotto certe condizioni di ricerca. Questa organizzazione è detta solitamente file hash. La condizione di ricerca deve essere una condizione di uguaglianza su un campo singolo, detto campo hash del file. Di solito il campo hash è anche un campo chiave del file, in questo caso è detto chiave hash. L'idea che sta dietro l'hash è quella di fornire una funzione h, detta funzione hash o funzione di randomizzazione, che è applicata al valore del campo hash di un record e fornisce l'indirizzo del blocco di disco in cui è memorizzato un record. Una ricerca del record all'interno del blocco può essere effettuata in un buffer in memoria centrale. Per la maggior parte dei record si ha bisogno solo di un accesso a un singolo blocco per recuperare quel record.
L'hash è usato anche come struttura di ricerca interna in un programma, ogni volta che a un gruppo di record si accedere esclusivamente usando il valore di un campo.

5.9.1 Hash interno

Per file interni, l'hash è tipicamente implementato come una tabella hash costruita usando un array di record.
Funzionamento: supponiamo che l'indice dell'array vari da 0 a m-1; si hanno , slot il cui indirizzo corrisponde all'indice dell'array. Si usa una hash function che trasforma il valore del campo hash in un numero intero compreso tra 0 ed m-1. Un esempio di funzione hash è:

$$h(k) = k \bmod m$$

La maggior parte delle funzioni di hashing non garantisce che a valori distinti del campo hash corrispondano indirizzi distinti, poiché l'insieme dei possibili valori del campo hash è molto più grande dello spazio degli indirizzi.
Quando l'indirizzo determinato dalla funzione di hashing in corrispondenza del valore del campo hash di un nuovo record coincide con quello già assegnato ad un altro record se genera una collisione.
In tal caso, bisogna determinare un'altra posizione nella quale inserire il nuovo record; il processo di ricerca di tale posizione è chiamato risoluzione delle collisioni.
Tecniche di risoluzione delle collisioni:

- Indirizzamento aperto: procedendo dalla posizione occupata specificata dall'indirizzo hash, il programma verifica in ordine le posizioni successive finché non si trova una posizione vuota
- Concatenamento: per questo metodo si tengono varie locazioni di overflow, di solito estendendo l'array con un certo numero di posizioni. Inoltre, esso aggiunge un campo puntatore a ogni posizione dell'array. Ogni collisione è risolta posizionando ilo nuovo record in una locazione di overflow non utilizzata e settando il puntatore dell'ultima posizione occupata incontrata all'indirizzo della locazione di overflow:

	Data fields	Overflow pointer
0		-1
1		M
2		-1
3		-1
4		M+2
●		
●		
●		
M-2		-1
M-1		M
M		-1
M+1		-1
M+2		M+2

- Hash multiplo: il programma applica una seconda funzione di hash e, in caso di seconda collisione continua con l'open addressing.

L'obiettivo di una funzione di hashing è quello di distribuire i record uniformemente nello spazio di indirizzi, in modo da minimizzare le collisioni e non lasciare troppe locazioni inutilizzate.
Simulazioni e studi hanno dimostrate che la soluzione ottimale è quella di avere una hash table completa al 70-90%, in modo che il numero di collisioni rimanga basso e al tempo stesso non si sprechi troppo spazio.
Ad esempio se immaginiamo di dover memorizzare r record nella tabella, dobbiamo utilizzare una spazio degli indirizzi comprendente m locazioni in modo che il rapporto r/m sia compreso tra 0.7 e 0.9.

5.9.2 Hash esterno per file su disco

La tecnica di hashing per file che risiedono su disco (disk file) è chiamata external hashing.

Date le caratteristiche di memorizzazione su disco, lo spazio degli indirizzi è composto da celle (bucket), ognuna delle quali contiene più record. Una cella può essere costituita sia da un singolo blocco che da un insieme di blocchi contigui.
Una tabella contenuta nel file header converte il numero di cella nel corrispondente indirizzo del blocco su disco. Il problema delle collisioni è meno grave rispetto all'internal hashing, poiché molti record possono essere contenuti nella stessa cella.
Insorgono dei problemi solo quando una cella è piena e un nuovo record deve essere inserito in quella cella. Il tal caso si usa una variante del chaining, in cui ogni cella mantiene un puntatore ad una lista di record di overflow per quella cella.

5.9.3 Tecniche di hash che consentono un'espansione dinamica dei file.

Esistono tre tecniche diverse che permettono di espandere dinamicamente i file:
Dynamic hashing
Il numero di celle non è fisso (come nel caso dell'hashing statico), ma può crescere o diminuire a seconda delle necessità.
Il file inizia con una sola cella. Quando questa cella è piena e un nuovo record deve venir inserito, la cella viene divisa in due. I record sono distribuiti tra le due celle sulla base del valore del primo bit (il più significativo) dei loro valori di hash. Tale processo viene iterate qualora una delle due celle risulti piena, utilizzando il secondo bit più significativo e così via.

Extendible hashing
È basato su un tipo di directory differente: un array di 2^d celle di indirizzi di celle di memoria.
L'esponente d è detto global depth della directory. Il valore intero corrispondente ai primi d bit più significativi del valore di hash è utilizzato come indice per l'accesso all'array/directory; nell'elemento dell'array così determinato è contenuto l'indirizzo della cella dove e memorizzato il record.
In generale, ad ognuna delle 2^d directory location non corrisponde una cella distinta: diverse directory location che coincidono con i primi d' (con d'<d) bit del loro hash value possono contenere lo stesso indirizzo; in tal caso, tutti i record che fanno riferimento a queste locazioni appartengono alla stessa cella.
Un global depth d può aumentare o diminuire, raddoppiando o dimezzando il numero di elementi dell'array/directory.

Linear Hashing
L'idea alla base del linear hashing è quella di permettere ad un file hash di aumentare e/o diminuire dinamicamente il numero di celle senza l'ausilio di alcuna directory.
Supponiamo che il file contenga inizialmente m celle, numerate da 0 a m-1, ed utilizzi una funzione di hash $h_o(k)$= k mod m. Gli overflow causati dalle collisioni sono inizialmente gestiti in modo standard, associando a ciascuna cella una catena di overflow. Quando però una nuova collisione genera una situazione in cui vi è almeno un record di overflow per ogni cella del file, la prima cella del file (cella 0) viene suddivisa in due celle: la cella originale 0 e una nuova cella m alla fine del file. I record originariamente contenuti nella cella 0 vengono distribuiti tra le due celle sulla base della nuova funzione di hashing: $h_1(k)$= k mod 2m. una proprietà delle due funzioni di hashing h_0 ed h_1 è che qualsiasi record che viene assegnato alla cella 0 dalla funzione h_0 è assegnato alla cella 0 o alla cella m dalla funzione h_1.
Le celle che sono state spezzate in due possono essere ricombinate qualora il carico del file (file load factor) cada al di sotto di una certa soglia. Si definisce file load factor il seguente valore l=r/(bfr*n) dove r è il numero corrente di record presenti nel file, bfr è il numero massimo di record che possono essere contenuti in una cella e n è il numero corrente di celle per file.

5.10 Altre organizzazioni primarie dei file

5.10.1 File di record misti

Le modalità di organizzazione dei file che abbiamo visto fino ad ora assumono che tutti i record di un particolare file siano dello stesso tipo, ma in molte applicazioni di DB, esistono numerose situazioni in cui diversi tipi di entità sono connessi in vari modi. In tali situazioni, può essere utile rappresentare le relazioni tra record di file diversi attraverso connecting field.
Ad esempio: un record Student può avere un connecting field MajorDept il cui valore fornisce il nome del Department in cui lo studente è majoring. Il campo MajorDept fa riferimento ad un'entità di tipo Department, rappresentata da record del file Department. Se si vogliono recuperare il valore di campi appartenenti a 2 record (di file diversi) tra loro collegati, occorre prima recuperare uno dei due record e poi usare il valore del suo connecting field per accedere al secondo record contenuto nell'altro file.
In tal modo, le relazioni sono implementate attraverso logical field references tra record di file distinti.
L'organizzazione dei file nelle basi di dati gerarchiche e reticolari implementa le relazioni tra record attraverso la physical relationship, che o sfruttano la contiguità fisica dei record coinvolti o utilizzano puntatori.
Questi modi di organizzare i file assegnano un'area del disco per memorizzare record di tipo diverso fra loro collegati in modo che possano essere fisicamente messi in relazione.
Ad esempio per gestire efficacemente interrogazioni che richiedano l'accesso ad un record di Department e a tutti i record di Student majoring in quel dipartimento, è utile memorizzare ogni record di department ed il relativo insieme di Student record in modo contiguo sul disco in un mixed file.
Per distinguere i record di un mixed file, ogni record ha associato, oltre al suo valore, un record type field che specifica il tipo di record.
Il record type field è generalmente il primo campo di ogni record ed è utilizzato dal sistema per determinare il tipo di record da processare.
Alcune strutture dati utilizzate per l'organizzazione dei file nel caso in cui sia la dimensione dei record che il numero dei record siano limitati sono i B-Tree.

6 Le strutture di indici per i file

Gli indici (indexes) sono strutture di accesso ai dati utilizzate per rendere più veloce il recupero di record in presenza di determinate condizioni di ricerca. Alcuni tipi di indici (secondary access paths) non influenzano la posizione fisica dei record sul disco, ma definiscono dei cammini di ricerca alternativi per localizzare efficientemente i record sulla base di opportuni campi indice (indexing fields). I più importanti tipi di indice sono basati su file ordinati (single-level indexes) e su strutture dati ad albero (multilevel indexes, B^+-Trees).

6.1 Tipi di indici ordinati a un solo livello

L'idea su cui si basa la struttura di accesso degli indici ordinati è simile a quella che sta alla base dell'indice analitico di un libro di testo: accanto ad ogni parola vi è la lista dei numeri di pagina dove la parola compare nel testo (alternativamente bisogna scorrere tutto il testo parola per parola: linear search).
Per un file con una data struttura di record che consiste di parecchi campi (o attributi), la struttura di accesso a indice è di solito definita in un unico campo, chiamato campo di indicizzazione (oppure attributo di indicizzazione), L'indice di solito memorizza ogni valore del campo di indicizzazione corredandolo di un elenco di puntatori a tutti i blocchi del disco che contengono record con quel valore del campo. I valori nell'indice sono ordinati in modo che si possa eseguire una ricerca binaria. Il file dell'indice è ovviamente molto più piccolo rispetto al file dei dati, quindi una ricerca

binaria è efficace. L'indicizzazione a più livelli evita la necessità di una ricerca binaria, ma richiede la creazione di ulteriori indici all'indice stesso.
Esistono molti tipi di indici ordinati. L'indice primario è specificato sul campo chiave di ordinamento di un file ordinato di record. Il campo chiave di ordinamento è utilizzato per ordinare fisicamente i record dei file su disco e tutti i record hanno un valore univoco per quel campo. Se il campo di ordinamento non è un campo chiave, cioè se molti record nel file possono avere lo stesso valore del campo di ordinamento, può essere usato un altro tipo di indice chiamato indice di cluster. Si noti che un file può avere al massimo un campo di ordinamento fisico, quindi può avere al massimo un indice primario o un indice di cluster, ma non entrambi. Su qualsiasi campo non di ordinamento di un file può essere specificato un terzo tipo di indice detto indice secondario. Un file può avere molti indici secondari.

6.1.1 indici primari

Un indice primario è un file ordinato i cui record sono di lunghezza fissa e sono costituiti da due campi. Il primo campo è dello stesso tipo di dati del campo chiave di ordinamento, chiamato chiave primaria, del file di dati e il secondo campo è un puntatore a un blocco del disco (un indirizzo di blocco), Esiste una voce (o record dell'indice) nel file dell'indice per ogni blocco nel file di dati. Ogni voce è composta da due campi che contengono il valore del campo della chiave primaria del primo record del blocco e un puntatore al corrispondente blocco su disco. Nel seguito, si farà riferimento ai due valori alla voce i come <K(i), P(i)>.
Il numero complessivo di voci dell'indice è uguale al numero di blocchi su disco nel file di dati ordinato. Il primo record di ciascun blocco del file di dati è chiamato record àncora del blocco o semplicemente punto àncora.
Gli indici di questo tipo possono essere densi o sparsi: un indice denso contiene una voce per ogni valore della chiave di ricerca (e quindi per ogni record) nel file di dati; un indice sparso contiene voci solo per alcuni valori di ricerca. L'indice primario è quindi un indice sparso, poiché include una voce per ogni blocco del file di dati piuttosto che per ogni valore di ricerca o per ogni record.
Esempio: supponiamo di avere un file ordinato contenente r=30˙000 record memorizzati su un disco con dimensioni del blocco B=1024 byte. I record diano di lunghezza fissa R=100 byte e unspanned. Il blocking factor per il file è

$$bfr = \lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10 \text{ record per blocco}$$

Il numero di blocchi necessari per memorizzare il file è

$$nb = \lceil r/bfr \rceil = \lceil 30.000/10 \rceil = 3000 \text{ blocchi}$$

Una ricerca binaria sul file di dati richiede approssimativamente

$$\lceil \log_2 nb \rceil = \lceil \log_2 3000 \rceil = 12 \text{ accessi al blocco}$$

Supponiamo ora che il campo contenere la ordering key del file sia lungo V=9 byte e che il puntatore ad un blocco sia lungo P=6 byte. La dimensione di ogni index entry R_i è:

$$R_i = 9+6 = 15 \text{ byte}$$

mentre il blocking factor è:

$$bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68 \text{ index entry per blocco.}$$

Il numero totale di index entry r_i è uguale al numero di blocchi del file, che è 3000. Il numero di blocchi necessario per l'indice è

$$nbi = \lceil r_i/bfr_i \rceil = \lceil 3000/68 \rceil = 45 \text{ blocchi}$$

Una ricerca binaria sul file indice richiede:

$$\lceil \log_2 nb_i \rceil = \lceil \log_2 45 \rceil = 6 \text{ accessi al blocco}$$

Conclusione: per cercare un record utilizzando l'indice occorrono 6 accessi a blocchi dell'indice più 1 accesso al blocco del file contenente il record, per un totale di 6+1=7 accessi, contro i 12 richiesti da una binary search (senza l'uso dell'indice).

Il problema principale degli indici primari (analogamente a quanto accadeva con i file ordinati) è quello dell'inserimento e della cancellazione di record, con in più l'onere di dover modificare anche

le index entry. Per ovviare a tale problema, si può utilizzare un file di overflow non ordinato, oppure aggiungere una lista di record di overflow a ogni blocco di dati.

6.1.2 Indici di cluster

Se i record di un file sono ordinati fisicamente ripestto a un campo che non è chiave, cioè che non ha un valore stinto per ciascun record, quel campo è chiamato campo di raggruppamento. È possibile creare un apposito indice, detto indice di cluster, per velocizzare il recupero dei record che hanno lo stesso valore del campo di raggruppamento. Questo indice differisce da un indice primario, che richiede che il campo di ordinamento del file di dati abbia un distinto valore per ogni record.
Anche l'indice di cluster è un file ordinato con due campi: il primo è del stesso tipo del campo di raggruppamento del file di dati, mentre il secondo è un puntatore ad un blocco. Vi è una voce nell'indice di cluster per ogni valore distinto del campo di raggruppamento, la quale contiene il valore del campo e un puntatore al primo blocco nel file di dati che contiene un record con quel valore del campo di raggruppamento.
Si noti che l'inserimento e l'eliminazione dei record causano ancora problemi perché i record dei dati sono fisicamente ordinati. Per ridurre il problema dell'inserimento è possibile riservare un intero blocco (oppure un raggruppamento di blocchi contigui) per ciascun valore del campo di raggruppamento; tutti i record con quel valore del campo sono posti nel blocco o nel raggruppamento di blocchi. Questo rende l'inserimento e l'eliminazione semplici.
Un indice di cluster è un altro esempio di indice sparso, perché contiene una voce per ogni valore distinto del campo di indicizzazione piuttosto che per ogni record nel file. La differenza principale con l'indice primario è che la ricerca tramite indice utilizza i valori del campo di ricerca, mentre la ricerca delle director hash usa un valore hash che è calcolato applicando la funzione hash al campo di ricerca.

6.1.3 Indici secondari

Un indice secondario è anch'esso un file ordinato con due campi. Il primo campo è dello stesso tipo di un campo del file di dati, che non viene utilizzato per effettuare l'ordinamento del file di dati, chiamato campo di indicizzazione. Il secondo capo è un puntatore a blocco oppure un puntatore a record. Vi possono essere più indici secondari (e quindi campi di indicizzazione) per lo stesso file.
Si prenda in considerazione prima di tutto una struttura di accesso del tipo indice secondario su un campo chiave che ha un valore distinto per ciascun record. Un campo di questo tipo è chiamato chiave secondaria. In questo caso vi è una voce dell'indice per ciascun record del file di dati, la quale contiene il valore della chiave secondaria del record e un puntatore al blocco in cui è memorizzato il record o al record stesso. Questo indice, quindi, è denso.
Indichiamo di nuovo i valori dei due campi di una voce i con <K(i), P(i)>. Le voci sono ordinate rispetto al valore K(i), quindi si può eseguire una ricerca binaria. Visto che i record del file di dati non sono fisicamente ordinati rispetto ai valori del campo della chiave secondaria, non è possibile utilizzare i punti àncora dei blocchi. Questo perché viene creata una voce dell'indice per ciascun record nel file di dati invece che per ogni blocco come nel caso di un indice primario.
Un indice secondario di solito ha bisogno di più spazio di memorizzazione e di un tempo di ricerca più lungo rispetto a un indice primario a causa del suo maggiore numero di voci. Il miglioramento nel tempo di ricerca di un record arbitrario, tuttavia, è decisamente maggiore per un indice secondario che per un indice primario, visto che se l'indice secondario non esistesse si dovrebbe svolgere una ricerca lineare nel file di dati. Nel caso dell'indice primario, invece, si potrebbe eseguire una ricerca binaria sul file principale, anche se l'indice non esistesse.
Esempio: si prenda in considerazione l'esempio di prima: r = 30.000 record di lunghezza fissa di dimensioni R = 100 byte memorizzati su un disco con dimensioni di blocco B = 1024 byte. Il file contiene 3000 blocchi. Per eseguire una ricerca lineare sul file in media sarebbero necessari b/2 = 3000/2=1500 accessi ai blocchi. Si supponga di creare un indice secondario su un campo chiave del

file, sul quale non è stato effettuato un ordinamento e che è lungo V=9 byte. Il puntatore ai blocchi è lungo P=6 byte, quindi ciascuna voce dell'indice è $R_i = (9+6) = 15$ byte e il fattore di blocco per l'indice è $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor 1024/15 \rfloor = 68$ voci per blocco. In un indice secondario denso come questo, il numero totale di voci dell'indice r_i è uguale al numero di record nel file di dati che è 30.000. Il numero di blocchi necessari per l'indice è quindi $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (30.000/68) \rceil = 442$ blocchi.
Una ricerca binaria su questo indice secondario richiede $\lceil \log_2 b_i \rceil = \lceil \log_2 442 \rceil = 9$ accessi ai blocchi. Per cercare un record usando l'indice è necessario un ulteriore accesso al blocco del file di dati per un totale di 9+1=10 accessi, un enorme miglioramento rispetto ai 1500 accessi mediamente necessari per una ricerca lineare, ma leggermente peggiore dei sette accessi necessari usando l'indice primario.
Si può creare anche un indice secondario su un campo non chiave di un file. In questo caso numerosi record nel file di dati possono avere lo stesso valore del campo di indicizzazione. Esistono varie tecniche per realizzare un indice di questo tipo:

1. Includere più index entry con lo stesso valore k(i), una per ogni record (indice denso)
2. Considerare record di lunghezza variabile per le index entry, con un campo (più volte) ripetuto per il puntatore. Viene mantenuta una lista di puntatori <p(i,1)…p(i,k)> nella index entry per k(i) (un puntatore ad ogni blocco che contiene un record il cui indexing field value è uguale a k(i)).
3. Il puntatore p(i) in una index entry <k(i), p(i)> punta ad un blocco; ogni puntatore presente nel blocco punta ad un record del file di dati con un valore k(i) dell'indexing field. Se i valori di k(i) sono condivisi da molti record (i relativi puntatori ai record non possono essere contenuti in un unico blocco), si utilizza una lista di blocchi.

6.2 Indice multilivello

Gli schemi d'indicizzazione descritti finora richiedono che il file dell'indice sia ordinato. Si usa poi la ricerca binaria per individuare i puntatori a un blocco sul disco oppure a un record nel file che ha il valore specificato nel campo d'indicizzazione. La ricerca binaria richiede approssimativamente $(\log_2 b_i)$ accessi ai blocchi per un indice con b_i blocchi perché ogni passo dell'algoritmo riduce la parte dell'indice in cui si deve continuare a cercare di un fattore pari a 2. L'idea che sta alla base della ricerca multilivello è ridurre a ogni passo la parte dell'indice in cui si deve continuare a cercare di un fattore pari a bfr_i, il fattore di blocco dell'indice, che è maggiore di 2. Lo spazio di ricerca, quindi, viene ridotto molto più velocemente. Il valore bfr_i è chiamato il fan-out dell'indice multilivello e si farà riferimento ad esso con il simbolo fo. La ricerca di un indice multilivello richiede approssimativamente $\log_{10} b_i$ accessi ai blocchi, che è un numero inferiore rispetto a quello della ricerca binaria se il fan-out è maggiore di 2.
Un indice multilivello considera il file dell'indice, al quale si fa riferimento ora come indice di primo livello (o livello base) di un indice multilivello, come un file ordinato con un valore distinto per ogni k(i). A questo punto è possibile creare un indice primario per l'indice di primo livello; questo indice dell'indice di primo livello è detto secondo livello dell'indice multilivello. Poiché il secondo livello è un indice primario, si possono utilizzare punti àncora dei blocchi in modo che il secondo livello contenga una voce per ciascun blocco del primo livello. Il fattore di blocco bfri per il secondo livello, e per tutti i livelli successivi, è lo stesso dell'indice di primo livello, perché tutte le voci dell'indice hanno la medesima dimensione; ciascuna ha un valore del campo e un indirizzo di blocco. Se il primo livello ha r_1 voci e il fattore di blocco, che è anche il fan-out, è bfr_i = fo, il primo livello necessita di $\lceil (r_1/fo) \rceil$ blocchi, che è quindi il numero di voci r_2 necessarie al secondo livello dell'indice.
Si può ripetere questo procedimento anche per il secondo livello. Il terzo livello, che è un indice primario per il secondo livello, ha una voce per ogni blocco del secondo livello; il numero di voci del terzo livello è quindi $r3 = \lceil (r_2/fo) \rceil$. Si noti che è richiesto un secondo livello solo se il primo

necessità di più blocchi di disco. Si può ripetere il procedimento precedente finché tutte le voci di un livello t dell'indice possono essere contenute in un singolo blocco. Questo blocco di livello t-esimo viene chiamato livello superiore dell'indice. Ogni livello riduce il numero di voci del livello precedente di fo, il fan-out dell'indice, cosicché è possibile usare la formula $l \leq (r_1/(fo)^t))$per calcolare t. Quindi un indice multilivello con r_1 voci del primo livello avrà approssimativamente t livelli, dove $t = \lceil (\log_{fo}(r_1)) \rceil$.
Esempio: Si supponga che l'indice denso dell'esempio precedente venga trasformato in un indice multilivello. Si è già calcolato il fattore di blocco $bfr_i = 68$ voci dell'indice per blocco, che è anche il fan-out fo per l'indice multilivello; è stato anche calcolato il numero di blocchi del primo livello b_1=442. Il numero di blocchi del secondo livello sarà $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ blocchi e il numero di blocchi del terzo livello sarà $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil 0$ 1 blocco. Per accedere a un record eseguendo una ricerca nell'indice multilivello, si deve accedere a un blocco di ciascun livello più al blocco del file di dati, quindi servono 3+1=4 accessi a blocchi.

6.3 Indici dinamici multilivello implementati da B-alberi e B$^+$-alberi

I B-alberi e i B$^+$-alberi sono casi speciali di strutture ad albero. Un albero è formato da nodi. Ogni nodo dell'albero, ad eccezione di un nodo speciale detto radice, ha un nodo padre e zero, uno o più nodi figli. Se non ha nessun nodo figlio è detto nodo foglia (leaf node). Ogni nodo che non sia un nodo foglia è detto nodo interno (internal node).
Un sottoalbero di un nodo comprende quel nodo e tutti i suoi discendenti.

6.3.1 Alberi di ricerca e B-alberi

Un albero di ricerca è uno speciale tipo di albero utilizzato per guidare la ricerca di un record, dato il valore di uno dei loro campi. Un indice multilivello può essere visto come una variante degli alberi di ricerca. Ogni nodo dell'indice multilivello può avere fo puntatori e fo chiavi. In ogni nodo, il valore del campo indice consente di passare al nodo successivo funchè non si raggiunge il blocco di dati che contiene il nodo cercato. Ad ogni livello, si restringe la ricerca di un sottoalbero radicato nel nodo corrente, ignorando tutti gli altri nodi dell'albero).
Un albero di ricerca di ordine p è un albero i cui nodi contengono al più p-1 search value e p puntatori nell'ordine:

$$<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$$

Con $q \leq p$. P_i è un puntatore ad un nodo figlio (o un puntatore nullo) e ogni K_i è un search value appartenente ad un insieme ordinato (assumiamo per semplicità che i search value K_i siano unici).
Ogni albero di ricerca deve soddisfare due vincoli fondamentali:

1. In ogni nodo $K_1 < K_2 < \ldots < K_{q-1}$
2. Per tutti i valori di X presenti nel sottoalbero puntato da P_i, vale la seguente relazione:

   $K_{i-1} < X < K_i$ per $1<i<q$
   $X < K_i$ per $i=1$
   $K_{i-1} < X$ per $i =q$

Un albero di ricerca può essere utilizzato per cercare record memorizzati su disco. I valori di ricerca (search value) possono essere i valori di uno dei campi dei file (search field).
Ad ogni valore di ricerca è associato un puntatore al record avente quel valore (oppure al blocco contenente quel record) nel file di dati.
L'albero stesso può essere memorizzato su disco, assegnando ad ogni nodo dell'albero un blocco.
Quando un nuovo record deve essere inserito, l'albero di ricerca deve essere aggiornato includendo il valore del campo di ricerca del uovo record ed il relativo puntatore al record nell'albero di ricerca.
Per inserire (o cancellare) valori di ricerca nell'albero di ricerca sono necessari specifici algoritmi che garantiscano il rispetto dei due vinxoli fondamentali. In generale tali algoritmi non garantiscono che un albero di ricerca risulti sempre bilanciato (nodi foglia tutti allo stesso livello).
Un albero di ricerca bilanciato evita che alcuni record richiedano molti più accessi a blocco di altri.

La cancellazione di record crea nell'albero nodi scarsamente popolati, aumentando la dimensione dello spazio inutilizzato e, conseguentemente, il numero di livelli.
Un B-Albero è un albero di ricerca con alcuni vincoli addizionali che garantiscono che l'albero sia bilanciato e lo spazio di memoria che diventa inutilizzato per effetto della cancellazione di record non diventi troppo grande.
Per soddisfare questi vincoli addizionali, occorrono algoritmi di inserzione e cancellazione più complessi, soprattutto nei casi di:

- Inserzione di un record in un nodo pieno
- Cancellazione di un record da un nodo che diventa pieno meno del 50% della sua dimensione

Formalmente, un B-albero di ordine p, se usato come struttura di accesso un campo chiave per la ricerca di record in un file di dati, deve soddisfare le seguenti condizioni:

1. Ogni nodo interno del B-albero ha la forma:
   $$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \dots , P_{q-1}, <K_{q-1}, Pr_{q-1}>P_q>$$
   Dove $q \leq p$, P_i è un tree pointer (puntatore ad un altro nodo del B-albero), Pr_i è un data pointer (puntatore ad un record il cui campo chiave di ricerca è uguale a K_i o al blocco che contiene tale record)
2. per ogni nodo, si ha che
   $$K_1 < K_2 < \dots < K_{q-1};$$
3. Ogni nodo ha al più p tree pointer
4. Per tutti i valori di X presenti nel sottoalbero puntato da P_i, vale la seguente relazione:
   $K_{i-1} < X < K_i$ per $1<i<q$
   $X < K_i$ per $i=1$
   $K_{i-1} < X$ per $i =q$
5. ogni nodo, esclusi la radice e i nodi foglia, ha almeno $\lceil p/2 \rceil$ tree pointers (la radice ha almeno 2 tree pointer, a meno che non sia l'unico nodo dell'albero).
6. un nodo con q tree pointer, $q \leq p$, ha q-1 campi chiave di ricerca (e, quindi, q-1 data pointer).
7. Tutti i nodi foglia sono posti allo stesso livello (i nodi foglia hanno la stessa struttura dei nodi interni, ad eccezione che tutti i loro tree pointer P_i sono nulli

Esempio 1: supponiamo che un campo di ricerca sia composto da V=9 byte, un blocco da B=512 byte, un record data pointer da P_r=7 byte e un block pointer da P = 6 byte.
Ogni nodo del B-albero può contenere al più p tree pointer e p-1 campi chiave di ricerca.
Tali puntatori possono essere tutti contenuti in un singolo blocco se ogni nodo del B albero corrisponde ad un blocco:

$$(p*P)+[(p-1)*(Pr+V)] \leq B$$
$$(p*6)+[(p-1)*(7+9)] \leq 512$$
$$22*p \leq 528$$

È ragionevole assegnare a p un valore elevato che soddisfi la disuguaglianza (ad esempio, p=23).
Esempio 2: supponiamo che il campo di ricerca dell'esempio precedente sia un nonordering key field, e si voglia costruire un B-albero su questo campo. Assumiamo, inoltre, che ogni nodo del B-albero sia pieno al 69%.
Ogni nodo conterrà (mediamente) p*0.69 = 23*0.69 = 16 puntatori (e, quindi, 15 valori del campo chiave di ricerca).
Il fan-out medio sarà fo=16
Partendo dal nodo radice, vediamo quanti valori e puntatori esistono mediamente ad ogni livello:

radice	1 nodo	15 entry	16 puntatori
Livello 1	16 nodo	240 entry	256 puntatori
Livello 2	256 nodo	3840 entry	4096 puntatori
Livello 3	4096 nodo	61440 entry	

Ad ogni livello è possibile determinare il numero di entry moltiplicando il numero totale di puntatori del livello precedente per 15 (il numero medio di entry per ogni nodo).
Date le dimensioni del blocco, la dimensione del puntatore e la dimensione del campo chiave di ricerca dell'esempio, un B-albero a due livelli avrà mediamente 3840+240+15 entry, mentre un B-albero su tre livelli avrà mediamente 65535 entry.

6.3.2 B^+-alberi

Un B^+-albero è un B-albero in cui i data pointer sono memorizzati solo nei nodi foglia dell'albero. La struttura dei nodi foglia differisce da quella dei nodi interni. Se il campo di ricerca è un campo chiave, nodi foglia hanno per ogni valore del campo di ricerca una entry ed un puntatore ad un record. Se un campo di ricerca non èun campo chiave, i putantori indirizzano un blocco che contiene puntatori ai record del file di dati, rendendo così necessario un passo aggiuntivo per l'accesso ai dati. I nodi foglia di un B^+-albero sono generalmente messi fra loro in relazione. Essi corrispondono al primo livello di un indice.
I nodi interni di un B^+-albero corrispondono al primo livello di un indice. I nodi interni di un B^+-albero corrispondono agli altri livelli di un insieme. Alcuni valori del campo di ricerca presenti nei nodi foglia sono ripetuti nei nodi interni per guidare la ricerca.
La struttura dei nodi interni di un B^+-albero di ordine p è la seguente:

1. Ogni nodo interno del B^+-albero ha la forma:
   $$<P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q>$$
   Dove $q \leq p$, P_i è un tree pointer (puntatore ad un altro nodo del B^+-albero)
2. per ogni nodo interno, si ha che
   $$K_1 < K_2 < \dots < K_{q-1};$$
3. Ogni nodo ha al più p tree pointer
4. Per tutti i valori di X della search key presenti nel sottoalbero puntato da P_i, vale la seguente relazione:
   $K_{i-1} < X \leq K_i$ per $1<i<q$
   $X \leq K_i$ per $i=1$
   $K_{i-1} < X$ per $i=q$
5. Ogni nodo, esclusi la radice e i nodi foglia, ha almeno $\lceil p/2 \rceil$ tree pointers (la radice ha almeno 2 tree pointer, a meno che non sia l'unico nodo dell'albero).
6. un nodo con q tree pointer, $q \leq p$, ha q-1 campi chiave di ricerca (e, quindi, q-1 data pointer).

La struttura dei nodi di un B^+-albero di ordine p è la seguente:

1. Ogni nodo foglia del B^+-albero ha la forma:
   $$<<K_1, Pr_1>, <K_2, Pr_2>, \dots, <K_{q-1}, Pr_{q-1}>P_{next}>$$
   Dove $q \leq p$, Pr_i è un data pointer e P_{next} punta al successivo nodo foglia del B^+-albero
2. per ogni nodo, si ha che
   $$K_1 < K_2 < \dots < K_{q-1};$$
3. Ogni Pr_i è un data pointer che punta a quel record il cui valore del campo di ricerca è K_i o ad un blocco contenente quel record
4. ogni nodo foglia ha almeno $\lceil p/2 \rceil$ tree pointers
5. Tutti i nodi foglia sono posti allo stesso livello

Per un B^+-albero costruito su un campo chiave, i puntatori appartenenti ai nodi interni sono tree pointer a blocchi che sono nodi dell'albero, mentre i puntatori appartenenti ai nodi foglia sono data pointer a record (o blocchi) del file di dati, ad eccezione di P_{next} che è un tree pointer al successivo nodo foglia.
Esempio: Per calcolare l'ordine p di un B^+-albero, supponiamo che il campo chiave di ricerca abbia dimensione V=9 byte, il blocco B=512 byte, il puntatore ad un record P_r=7 byte e il puntatore ad un blocco P=6 byte.

Un nodo interno di un B^+-albero può avere fino a p tree pointer e p-1 valori del campo di ricerca (che devono essere contenuti in un singolo blocco):

$$(p*P)+[(p-1)*V] \leq B$$
$$(p*6)+[(p-1)*9] \leq 512$$
$$(p*15) \leq 528$$

Si sceglie come valore di p il più alto valore che soddisfa la disuguaglianza, ossia p=34. Questo valore è più alto di quello del corrispondente B-Albero (p=23): si ha un valore di fan-out più elevato e un numero maggiore di entry in ogni nodo interno rispetto al corrispondente B-Albero. I nodi foglia del B^+-albero hanno lo stesso numero di valori e di puntatori, eccetto che i puntatori sono data-pointers e next-pointers.
L'ordine p_{leaf} per i nodi foglia può essere calcolato come:

$$[p_{leaf} * (P_r + V)] + P \leq B$$
$$[p_{leaf} * (7+9)] + 6 \leq 512$$
$$16*p_{leaf} \leq 506$$
$$p_{leaf} = 31$$

Esempio: supponiamo di voler costruire un B^+-albero sul campo dell'esempio precedente.
Assumiamo che ogni nodo del B^+-albero sia pieno al 69%.
Ogni nodo interno mediamente avrà p*0.69 = 34*0.69 = 23 puntatori, e quindi 22 valori.
Ogni nodo foglia mediamente avrà p_{leaf}*0.69 = 31*0.69 = 21 puntatori a record di dati.
Un B^+-albero avrà il seguente numero medio di entry ad ogni livello:

radice	1 nodo	22 entry	23 puntatori
Livello 1	23 nodo	506 entry	529 puntatori
Livello 2	529 nodo	11638 entry	12167 puntatori
Livello 3	12167 nodo	255507 entry	

Fissate le dimensioni del blocco, la dimensione del puntatore e la dimensione del campo chiave di ricerca, un B^+-albero di tre livelli avrà 255507 entry contro le 65535 di un B-albero.

Indice

PARTE 1. Concetti di Base. 2
PARTE 2. Il modello Entità/Relazioni (ER). 2
PARTE 3. Il Modello Relazionale. 2
PARTE 4. La teoria della progettazione delle basi di dati relazionali. 2
PARTE 5. L'Organizzazione Fisica dei Dati. 2
1.1 Introduzione 3
1.3 Caratteristiche dell'approccio con BdD 4
1.3.1 Natura autodescrittiva di un sistema di basi di dati 4
1.3.2 Separazione tra programmi e dati e astrazione dei dati 4
1.3.3 Supporto di viste multiple dei dati 5
1.3.4 Condivisione dei dati ed elaborazione delle transazioni con utenti multipli. 5
1.4 Gli attori in scena 5
1.4.1 Il gestore di una base di dati 5
1.4.2 l'amministratore di una base di dati 5
1.4.3 Utenti di una base di dati 5
2.1 Modelli di dati, schemi e istanze 6
2.1.1 Categorie di modelli di dati 6
2.1.2 Schemi, istanze e stato di una BdD 7
2.2 Architettura di un DBMS e indipendenza dei dati. 8
2.2.1 L'architettura a tre livelli 8
2.2.2 Indipendenza dei dati 8
2.3 Linguaggi e interfacce di BdD 9
2.3.1 Linguaggi dei DBMS 9
7. Il modello di dati, i vincoli e l'algebra relazionali 10
7.1 Concetti del mondo relazionale 10
7.1.1 Domini, attributi, tuple e relazioni. 10
7.1.2 Caratteristiche delle relazioni 11
7.1.3 Notazione del modello relazionale 11
7.2 Vincoli relazioni e schemi di BdD relazionali 11
7.2.1 Vincoli sul dominio 11
7.2.2 Vincoli di chiave e vincoli sui valori nulli. 11
7.2.3 BdD e schemi di BdD relazionali 12
7.2.4 Integrità dell'entità, integrità referenziale e chiavi esterne 12
7.3 Operazioni di aggiornamento e gestione delle violazioni dei vincoli 12
7.3.1 L'operazione di inserimento 12
7.3.2 L'operazione di cancellazione 13
7.3.3 L'operazione di modifica 13
7.4 Operazioni di base dell'algebra relazionale 13
7.4.1 L'operatore di selezione. 13
7.4.2 L'operazione di proiezione 14
7.4.3 Sequenze di operazioni e operazione di ridenominazione 14
7.4.4 Operazioni insiemistiche 15
7.4.5 L'operazione di join 16
7.4.6 Un insieme completo di operazioni dell'algebra relazionale. 16
7.4.7 L'operazione di divisione 16
7.5 Altre operazioni relazionali 17
7.5.1 Funzioni aggregate e raggruppamento 17
7.5.2 Operazioni di chiusura ricorsiva 17

7.5.3 Operazioni di join esterno e di unione esterna 18
8.1 La definizione dei dati e le modifiche agli schemi in SQL2 18
8.1.1 I concetti di schema e di catalogo 18
8.1.2 Il comando create table e i vincoli di tipi di dati 19
8.1.3 I comandi DROP SCHEMA e DROP TABLE. 20
8.1.4 Il comando ALTER TABLE 20
8.2 Interrogazioni fondamentali in SQL2 20
8.2.1 La struttura SELECT – FROM – WHERE delle interrogazioni SQL 20
8.2.2 La problematica dei nomi di attributi ambigui e la loro ridenominazione (assegnando pseudonimi) 21
8.2.3 La clausola WHERE non specificata e l'uso dell'asterisco (*) 22
8.2.4 Le tabelle come insiemi in SQL 22
8.2.5 Confronti di sottostringhe, operatori aritmetici e ordinamento 23
8.3 Interrogazioni SQL più complesse 25
8.3.1 Interrogazioni nidificate e confronti di insiemi 25
8.3.2 Le funzioni EXISTS e UNIQUE in SQL 27
8.3.3 Insiemi espliciti e NULL 28
8.3.4 Ridenominazione degli attributi e delle tabelle collegate con i join 28
8.3.5 Funzioni di aggregazione e raggruppamento 29
8.4 Le istruzioni Insert, Delete, and Update in SQL 33
8.4.1 Il comando INSERT 33
8.4.2 Il comando DELETE 33
8.4.3 Il comando UPDATE 33
8.5 Viste (tabelle virtuali) in SQL 34
8.5.1 Il concetto di vista in SQL 34
8.5.2 Specificazione delle viste 34
10 Dipendenze funzionali e normalizzazione per basi di dati relazionali 36
10.1 Linee guida informali di progettazione di schemi di relazione 36
10.1.1 Semantica degli attributi di una relazione 36
10.1.2 Informazioni ridondanti nelle tuple e anomalie di aggiornamento 36
10.1.3 Valori nulli nelle tuple 37
10.1.4 Generazione di tuple spurie 37
10.2 Dipendenze funzionali 37
10.2.1 Definizione di dipendenza funzionale 37
10.2.2 Regole di inferenza per dipendenze funzionali 38
10.2.3 Equivalenza di insiemi di dipendenze funzionali 39
10.2.4 Insiemi minimali di dipendenze funzionali 39
10.3 Forme normali basate su chiavi primarie 40
10.3.1 Introduzione alla normalizzazione 40
10.3.2 Prima forma normale 41
10.3.3 Seconda forma normale 42
10.3.4 Terza forma normale 43
10.5 Forma normale di Boyce e Codd 44
5.1 Introduzione 45
5.1.2 Memorizzazione di basi di dati. 45
5.4 Bufferizzazione dei blocchi 46
5.5 Collocazione su disco dei record di un file 46
5.5.1 Record e tipi di record 46
5.5.2 File, record a lunghezza fissa e record a lunghezza variabile 46
5.5.3 Ripartizione dei record in blocchi e confronto tra record con spanning e record senza spanning 47

5.5.4 Allocazione dei blocchi di un file su disco 47
5.5.5 Header dei file 47
5.6 operazioni sui file. 48
5.7 File di record non ordinati (file heap) 49
5.8 File di record ordinati (file sorted) 49
5.9 Tecniche hash 50
5.9.1 Hash interno 50
5.9.2 Hash esterno per file su disco 51
5.9.3 Tecniche di hash che consentono un'espansione dinamica dei file. 52
5.10 Altre organizzazioni primarie dei file 53
5.10.1 File di record misti 53
6 Le strutture di indici per i file 53
6.1 Tipi di indici ordinati a un solo livello 53
6.1.1 indici primari 54
6.1.2 Indici di cluster 55
6.1.3 Indici secondari 55
6.2 Indice multilivello 56
6.3 Indici dinamici multilivello implementati da B-alberi e B^+-alberi 57
6.3.1 Alberi di ricerca e B-alberi 57
6.3.2 B^+-alberi 59
Indice 61